Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Magnus Midtbø Kristiansen

Proving Theorems Using Deep Learning

Graph Convolutional Networks, Transformers, and Deep Reinforcement Learning for Automatic Formal Reasoning

Master's thesis in Computer Science
Supervisor: Björn Gambäck
June 2021

NTNU
Norwegian University of Science and Technology

Magnus Midtbø Kristiansen

Proving Theorems Using Deep Learning

Graph Convolutional Networks, Transformers, and Deep Reinforcement Learning for Automatic Formal Reasoning

Master's thesis in Computer Science
Supervisor: Björn Gambäck
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Interactive Theorem Proving (ITP) systems are symbolic-based software systems used to write and verify formal mathematical proofs. These systems often contain large datasets of human-written formalized proofs, structured as step-by-step applications of high-level proof strategies called *tactics*. The space of tactics is well-defined and contains a combination of core tactics and *tactic arguments*. Tactic arguments generally refer to either already proven theorems or terms and hypotheses in the local proof search context. Recently, several research groups have focused on automating ITP systems by training machine learning models to predict what next tactic to apply in any given proof state. This has resulted in whole frameworks developed for more accessible research into machine learning models automating underlying ITP systems. Such ITP automation allows the model to perform high-level formal reasoning similar to human mathematical reasoning.

This Master's Thesis develops a new theorem proving agent for end-to-end ITP theorem proving. The agent transforms the theorem proving task into three separate multi-class classification problems, allowing a more natural machine learning interpretation of the theorem proving task than previous approaches.

In addition to models imitating human proofs via supervised learning, deep reinforcement learning – implemented using deep Q-learning – is deployed. This has two advantages: (1) it deals with data scarcity, and (2) it allows the agent to develop its own proof style, effectively circumventing noisy human-written proofs. Furthermore, two novel deep learning embedding techniques are tested: Graph Convolutional Networks (GCNs) and the Bidirectional Encoder Representations from Transformers (BERT) architecture. More general non-convolutional Graph Neural Networks have recently been shown to work well on formal logic and been used successfully for ITP theorem proving. BERT has shown state-of-the-art results on several Natural Language Processing tasks. In addition, Transformer-based models have recently shown promising results on related mathematical reasoning tasks.

When trained to imitate human proofs, GCN and BERT-based agents significantly outperform corresponding random guessing agents, proving 37.3% and 16.3% more theorems, respectively. Deep reinforcement learning improves results further. These agents are capable of proving 7.6% more theorems than corresponding supervised agents and 47.7% more theorems than corresponding random guessing agents. This is the first time GCN, Transformers, and deep reinforcement learning have been used for tactic-based formal theorem proving.

Sammendrag

Interaktive teorembevissystemer (ITP-systemer) er symbolbaserte programvaresystemer som brukes til å skrive og verifisere formelle matematiske bevis. Disse systemene inneholder ofte store datasett med menneskeskrevne formaliserte bevis, strukturert som trinnvise applikasjoner av bevis-strategier kalt *taktikker*. Rommet av mulige taktikker er veldefinert og inneholder en kombinasjon av kjernetaktikker og *taktikkargumenter*. Taktikkargumenter refererer som regel enten til allerede beviste teoremer eller termer og hypoteser i den lokale beviskonteksten. Nylig har flere forskningsgrupper fokusert på automatisering av ITP-systemer ved å trene maskinlæringsmodeller til å forutsi hvilken neste taktikk som skal brukes i bevissøket. Dette har resultert i rammeverk utviklet for mer tilgjengelig forskning på maskinlæringsmodeller som automatiserer underliggende ITP-systemer. En slik automatisering av ITP-systemer lar modeller utføre formell resonnering på et høyt abstraksjonsnivå, lignende menneskelig matematisk resonnering.

Denne masteroppgaven utvikler en ny bevisagent for ende-til-ende bevissøk i ITP-systemer. Agenten transformerer bevisproblemet til tre separate klassifiseringsproblemer, noe som gir en mer naturlig maskinlæringstolkning av bevisproblemet enn tidligere tilnærminger.

I tillegg til modeller som imiterer mennesskrevne bevis via veiledet læring, anvendes også dyp forsterkningslæring – implementert ved hjelp av dyp Q-læring. Dette har to fordeler: (1) det håndterer knappheten av annotert data, og (2) agenten har muligheten til å utvikle sin egen bevisstrategi, noe som lar den omgå støy i menneskeskrevne bevis. Videre testes to nye dyplæringsteknikker: Konvolusjonelle nevrale nettverk for grafstrukturer (GCNs) og Bidirectional Encoder Representations from Transformers (BERT) arkitekturen. Mer generelle ikke-konvolusjonelle nevrale nettverk for grafstrukturer er nylig vist å fungere godt på formell logikk og blitt brukt til å bevise teoremer i ITP-systemer. BERT har vist overlegne resultater på flere problemer innen språkbehandlings-feltet (Natural Language Processing). I tillegg har andre Transformer-modeller nylig vist lovende resultater på relaterte problemer innen formell logikk.

GCN- og BERT-baserte agenter beviser henholdsvis 37,3 % og 16,3 % flere teoremer enn tilsvarende agenter basert på tilfeldig gjetting, når de blir trent til å imitere menneskeskrevne bevis. Dyp forsterkningslæring forbedrer resultatene ytterligere. Disse agentene er i stand til å bevise 7,6 % flere teoremer enn tilsvarende veiledete agenter og 47,7 % flere teoremer enn tilsvarende agenter basert på tilfeldig gjetting. Dette er første gang GCN, Transformers og dyp forsterkningslæring er brukt til å automatisere taktisk-baserte ITP-systemer.

Preface

This Master's Thesis is written as part of the degree Master of Science in Computer Science at the Norwegian University of Science and Technology, under the supervision of Björn Gambäck. A special thanks goes out to Björn Gambäck for his valuable guidance and feedback throughout the entire duration of the project. A thanks also goes out to Kaiyu Yang at Princeton University for his helpful responses on the CoqGym discussion board. Furthermore, a thanks goes out to Felix Wu and Yixin Chen for allowing their figures to be depicted in the Thesis. The HPC group at NTNU also deserves a big thanks for allowing the use of the Idun cluster to conduct experiments.

I would also like to thank friends and family for their great support along the way. Finally, I would like to give a special thanks to Elise for having the patience to listen to my somewhat long-winded monologues about topics in this Thesis and for her continued support.

Magnus Midtbø Kristiansen
Trondheim, June 11, 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Automated Theorem Proving (ATP) is a field of study concerned with automatically proving mathematical theorems using a computer. Traditionally, a set of theorems and a conjecture (the theorem to be proven) are expressed formally, based on some logical framework, with the task of proving the conjecture focused around symbolic manipulation on the set of logically expressed statements. Even with state-of-the-art inference techniques deployed, this essentially turns into a combinatorial search problem, where one quickly encounters an exponentially increasing space of combinations (Hoder and Voronkov, 2011). In addition, validity in First-Order Logic, the most common logic used in ATP systems, is known to be a semi-decidable problem (Church, 1936; Turing, 1936). Because ATP systems often seek to prove validity, this means there is no effective way to disprove a conjecture that is in fact false.

Because of these issues, the field of Interactive Theorem Proving (ITP)[1] has emerged as an alternative way of doing computer-based theorem proving (Harrison et al., 2014). This branch of computer theorem proving is not concerned with a fully automated process, but instead tries to facilitate an enhanced theorem proving process for human users. This is made possible by letting the system deal with the tedious details of the proof, while the user guides the proof search by inputting high-level proof strategies (most commonly taking the form of so-called *tactics*). As with ATP systems, ITP systems are based on formal logic and designed to guarantee correctness of the produced proofs (Harrison et al., 2014). Such systems have become the de facto tools in efforts to formalize mathematics (Hales, 2006; Gonthier et al., 2013; Leroy, 2016).

Several machine learning researchers have recently used ITP systems as a way to tackle the domain of mathematics and formal reasoning (Bansal et al., 2019a; Huang et al., 2019; Yang and Deng, 2019; Gauthier et al., 2017). The main idea is to train machine learning models to predict the next tactic to apply and drive the ITP's proof procedure forward automatically. Because this approach automates an underlying ITP system, it will be referred to as *Auto-ITP* in this Thesis, a term coined by Yang and Deng (2019)[2]. In a way, Auto-ITP can be seen as a form of ATP. However, it very

[1] ITP systems are often also called *Proof Assistants*. However, this Thesis will only use the term ITP systems.

[2] Although the term "Auto-ITP" is not a widely adopted one, it does provide a useful shorthand term to refer to machine learning-driven automation of ITP systems.

different from the classical low-level inference techniques traditional ATP systems rely on. Instead, Auto-ITP emulates how a human user would interact with ITP systems. It can therefore be considered a more human-inspired approach to ATP than traditional ATP systems, where the model proves theorems on abstraction levels closer to human mathematical reasoning (Yang and Deng, 2019). The Auto-ITP process is similar to an *active learning* setup (Settles, 2009), where the Auto-ITP model serves as an (automatic) *oracle* for the underlying ITP system. The ITP system queries subgoals to the Auto-ITP model and the model responds with a tactic corresponding to the subgoal.

In the last couple of years, several frameworks for doing Auto-ITP have emerged (Bansal et al., 2019a; Huang et al., 2019; Yang and Deng, 2019; Wu et al., 2020). These frameworks allow machine learning researchers interested in the domain of mathematics to leverage powerful underlying ITP systems and large datasets of human-written proofs in the quest to progress machine learning applied to formal reasoning and the progress of artificial intelligence more broadly (Urban and Vyskočil, 2013; Szegedy, 2020).

This Master's Thesis will cover state-of-the-art within each existing Auto-ITP framework, in addition to other work related to Auto-ITP. This includes other applications of machine learning in mathematics and formal reasoning as well as another popular approach for automating underlying ITP systems – so-called *Hammer* systems (Blanchette et al., 2016). Then, the Thesis narrows its focus to a single Auto-ITP framework, in which new experiments will be conducted. The framework chosen is the *CoqGym* framework (Yang and Deng, 2019), with the overall goal to explore machine learning techniques not yet tested in CoqGym. This Master's Thesis tests new deep learning methods – based on Graph Convolutional Networks (GCNs) and the Bidirectional Encoder Representations from Transformers (BERT) model (Devlin et al., 2018) – as embedding techniques for Coq expressions. Models are trained both to imitate human proofs using supervised learning, and with the deep reinforcement learning method deep Q-learning (Mnih et al., 2015). In addition, a new theorem proving agent is developed, interpreting the ITP theorem proving process as three multi-class classification problems. Lastly, a proxy metric is designed to allow for less expensive prototyping of supervised learning models in CoqGym.

1.1 Background and Motivation

ATP is in itself motivated by several things. The most obvious might be the goal of proving new mathematical theorems. Some theorems lend themselves naturally to the formal way in which traditional ATP systems work, and in those cases ATP systems have performed reasonably well. An example of this is Robbins' problem, which asks if all Robbins algebras are Boolean algebras. This was proven by the EQP system in 1997 (McCune, 1997), essentially by brute-force calculations on combinations of First-Order expressions.

Another important use of both ATP and ITP systems is formal (and guaranteed correct) verification of logically expressed statements. This has been particularly useful in software and hardware verification, where behavior can be naturally expressed through formal logic. State-of-the-art systems have been used to verify the correctness of processors (Harrison, 2000), operating systems (Klein et al., 2014; Chen et al., 2015) and compilers (Leroy, 2009). Intel, for example, hired ITP pioneer John Harrison to verify floating point arithmetic on their processors. He developed the ITP system HOL Light (Harrison, 1996), capable of producing guaranteed correct verification of processor operations (Harrison, 2000).

Computer systems also provide a natural tool for formalizing mathematics. It has been a long-standing dream of computer scientists and mathematicians to one day formalize all of mathematics and science in a machine-understandable way – effectively reducing the problem of reasoning to "number crunching", which can be executed by a machine. This was made explicit in the *QED manifesto* (Boyer, 1994). While the QED vision has yet to come into full fruition, plenty of efforts in both ATP and ITP research aim to formalize mathematical proofs (Gonthier, 2008; Gonthier et al., 2013; Hales et al., 2017).

Auto-ITP, on the other hand, has its roots in the machine learning community and is consequently motivated by mathematics and formal reasoning being a challenging and relatively unexplored domain for machine learning models (Urban and Vyskočil, 2013; Kaliszyk et al., 2017; Szegedy, 2020). As pointed out by Urban and Vyskočil (2013) and Szegedy (2020), the theorem proving domain can potentially be used to develop new and novel machine learning methods. Because of the bridge between formal theorem proving and software systems, Auto-ITP can also be motivated as a steppingstone for developing models capable of software synthesis (Szegedy, 2020). An essential aspect of Auto-ITP research has been large datasets of proof data resulting from already completed large-scale formalization projects (Kaliszyk et al., 2017). This access to data opens up the door for data-hungry machine learning methods. However, frameworks and benchmarks have been lacking from the domain up until recently. Bansal et al. (2019a) argue that widely adopted benchmarks in other domains, such as ImageNet (Deng et al., 2009) for object detection and LibriSpeech (Panayotov et al., 2015) for speech recognition, have been instrumental for the success of machine learning in these domains. This has led to efforts by several research groups to provide both frameworks and benchmarks in the domain of theorem proving. These frameworks have mainly been developed with the goal of tactic application as a machine learning problem in mind, which has evolved into Auto-ITP.

It is also worth noting that the rapid progress machine learning (in particular deep learning) has experienced in the last few years motivates Auto-ITP from a traditional formal theorem proving perspective as well. In particular, large-scale

formalization work is massively labor-intensive[3] and more automation of such tasks is therefore desirable. This has been a significant motivation for developing Hammer systems, which are capable of proving large chunks of formalization projects automatically (Blanchette et al., 2016).

Experiments in this Thesis are primarily motivated by asking the following question: "What successes in machine learning (in the field of formal reasoning and at large) can be drawn on to explore new techniques for the Auto-ITP task?". In particular, this Thesis explores Graph Neural Networks and Transformer networks further, as they have recently shown promising results as embedding techniques for mathematical expressions (Paliwal et al., 2020; Rabe et al., 2020; Lample and Charton, 2020; Polu and Sutskever, 2020). Allowing models to train on not only human-written proofs but also machine-generated proofs has shown to improve results in the theorem proving context (Bansal et al., 2019b). Data scarcity is also a concern when training theorem proving models (Wang and Deng, 2020). This motivates experiments involving both supervised learning models and reinforcement learning models. Reinforcement learning allows the agent to learn from exploring tactic applications rather than from a curriculum.

Since there is overhead associated with achieving complete end-to-end theorem proving, the prototyping of new machine learning models can benefit from simplified proxy metrics indicating the prototype's success. However, because of the infancy of Auto-ITP frameworks, there is a lack of such methodology in the field. This motivates the focus on developing a proxy metric allowing easier prototyping of models. The development of a new theorem proving agent for ITP is motivated by a similar idea. Namely, it is helpful to have familiar machine learning interpretations of the theorem proving task when studying Auto-ITP.

1.2 Goals and Research Questions

Based on the backdrop described above, a single overarching Goal is formulated for this Master's Thesis:

Goal *Further progress machine learning applied to formal reasoning by testing new machine learning techniques on the Auto-ITP task.*

This Goal is fairly broad and could potentially encompass a vast amount of experiments. It will therefore be necessary to restrict the scope to a manageable set of ideas. The first is to restrict experiments to a single Auto-ITP framework. The chosen framework for this Thesis is the CoqGym framework (Yang et al., 2016), based on the popular ITP system Coq (Barras et al., 1997). Furthermore, four main ideas are pursued: (1) proxy metrics for Auto-ITP, allowing easy prototyping of models, (2) an end-to-end theorem proving agent easy to interpret from a machine learning perspective, (3) supervised

[3] An example of this is the formal proof of the Kepler conjecture (Hales et al., 2017), which took 20-person years to complete.

learning using novel embedding techniques and (4) reinforcement learning. Section 4.1 (after related work has been presented in Chapter 3) covers a more detailed explanation for why CoqGym and these ideas were chosen. The following Research Questions make the ideas more explicit:

Research Question 1 *How to design an easy and fast Auto-ITP proxy metric that also indicates end-to-end theorem proving performance?*

Auto-ITP is a domain where training and testing can be reasonably complicated and slow. Research Question 1 addresses the need for easier and faster prototyping of Auto-ITP models.

Research Question 2 *How can a conceptually simple end-to-end theorem proving agent be designed for tactic-based ITP theorem proving?*

Although tactic prediction is a fairly straightforward machine learning problem, it gets more complicated when tactic arguments are introduced. It is not clear exactly how to interpret ITP theorem proving as a machine learning problem. Thus, this Thesis argues that designing an agent where the theorem proving task is broken down into familiar machine learning problems can be helpful. Research Question 2 targets this topic.

Research Question 3 *What novel embedding techniques can help models perform well in CoqGym?*

Research Question 3 is based on the idea that the semantic information contained in logical expressions is likely essential for Auto-ITP models. Strong embedding networks have been hugely successful in Natural Language Processing (NLP), with Transformers like BERT (Devlin et al., 2018) becoming household names in this field. Similar attention networks have shown promising results for embedding mathematical expressions (Rabe et al., 2020; Lample and Charton, 2020; Polu and Sutskever, 2020), although not yet been used specifically for Auto-ITP. Embedding using Graph Neural Networks and TreeLSTM has already shown promising results in the Auto-ITP domain (Paliwal et al., 2020; Yang and Deng, 2019), and is therefore also an interesting approach to pursue further in CoqGym.

Research Question 4 *How does reinforcement learning compare to supervised learning in CoqGym?*

It might be the case that human-written proofs are noisy and hard to learn from. The proofs are gathered from different formalization projects, where different teams of humans have been involved. Bansal et al. (2019b) experienced significant improvements with their Auto-ITP model by letting the model learn from its own proofs and not only

human-written proofs. In other words, a machine learning model might be better off learning its own "style" of proving theorems rather than trying to imitate a human. Data scarcity has also been pointed out as a bottleneck for formal theorem proving models (Wang and Deng, 2020). Research Question 4 addresses these points in the context of CoqGym.

1.3 Research Method

A literature review of machine learning applied to formal reasoning is the starting point for this Master's Thesis research. A review of the mechanics of Coq and tactic-based ITP theorem proving in general is needed to answer Research Questions 1 and 2. Dataset statistics from CoqGym will also be used to answer Research Question 1 and 2. Research Questions 3 and 4 are answered using an experimental approach. Each model and theorem proving agent is compared against each other and to related results from the literature. To better understand the agents' performance, a random guessing baseline agent will be developed and tested.

1.4 Contributions

The main contributions of this Master's Thesis are the following:

1. *An Auto-ITP proxy metric based on predicting tactic groups.*
2. *An end-to-end theorem proving agent based on solving three multi-class classification problems.*
3. *Experiments using supervised Graph Neural Network – more specifically, Graph Convolutional Network – models in CoqGym. These models are trained on both human-written and synthetic proof steps.*
4. *Experiments using supervised BERT models in CoqGym. These models are trained on both human-written and synthetic proof steps. Models with and without pre-trained weights are trained and compared.*
5. *Experiments with end-to-end theorem proving agents, combining different Graph Convolutional Network models and BERT models.*
6. *Experiments with end-to-end theorem proving agents, trained using a combination of deep reinforcement learning and supervised learning.*
7. *Experiments with end-to-end theorem proving agents, with different depth limits and beam widths.*

All agents prove significantly more theorems than corresponding random guessing agents – 16.30% more for the BERT-based agent, 37.28% more for the Graph Convolutional

Network-based agent, and 47.76% more for the deep reinforcement learning agent. However, no agent is capable of outperforming state-of-the-art (First et al., 2020), with the best-performing agent proving 10.74% of the CoqGym test set – 2.16 percentage points lower than state-of-the-art. Note that a direct comparison to state-of-the-art is difficult as the theorem agent in this Thesis operates (by design) differently than the agent used by First et al. (2020).

1.5 Thesis Structure

Chapter 2 covers all relevant background theory necessary to follow the rest of the Thesis. This includes an introduction of traditional ATP and ITP, as well as relevant machine learning theory.

Chapter 3 covers relevant work. This includes work in Auto-ITP, Hammer systems and other applications of machine learning in the domain of mathematics and formal reasoning.

Chapter 4 explains the motivation for the experiments pursued in this Thesis. This chapter also explains how the proxy metric and the end-to-end theorem proving agent are designed, as well as the overall deep learning architectures used in the experiments.

Chapter 5 covers experiments and results. This includes concrete model configurations, the experimental setup, and a detailed account of the results.

Chapter 6 evaluates and discusses how the Master's Thesis has answered the Goal and Research Questions, in addition to further discussing findings from the experiments and relevant topics in this Thesis.

Finally, **Chapter 7** summarizes the Thesis' contributions and addresses possible future avenues of work.

Chapter 2

Background Theory

This chapter contains the necessary background theory needed to follow the experiments in this Thesis. In addition, some concepts are included to understand better the related work covered in Chapter 3. First, Section 2.1 covers a general introduction to traditional Automated Theorem Proving (ATP) systems. The focus is mainly on the inference techniques used by such systems. Although this section is not strictly needed to understand Auto-ITP, it is included because it provides more context to Interactive Theorem Proving (ITP) systems and, therefore, also Auto-ITP. In addition, Hammer systems, which are part of the body of related work in Chapter 3, rely on ATP systems, and many of the ATP inference techniques are used internally by ITP systems. Section 2.2 introduces the main ideas of traditional ITP systems. Relevant details on Coq (Chlipala, 2013) (CoqGym's underlying ITP system) are included in this section. The focus then shifts to machine learning, with Section 2.3 covering relevant machine learning theory.

It is assumed that the reader is already familiar with First-Order Logic, calculus, linear algebra, and statistics. ITP systems are typically based on Higher-Order Logic, but this topic is not strictly necessary for this Master's Thesis and is not included in the background theory. Väänänen (2020) provides an excellent introduction to Higher-Order Logic for the interested reader.

2.1 Traditional Automated Theorem Proving

An Automated Theorem Proving (ATP) system is a computer program operating within some logical framework (Bibel, 2007). This section focuses on the most common type of ATP system: systems based on First-Order Logic with equality (Schulz, 2002; Kovács and Voronkov, 2013). First-Order Logic's popularity stems from the fact that a vast amount of mathematics can be expressed formally through First-Order Logic (Ewald, 2019), while it at the same time offers fast inference techniques (some of which will be discussed here).

The general setup for an ATP system consists of a Knowledge Base of already known theorems, plus a new theorem to be proven. The new theorem is referred to as the *conjecture*. The system tries to infer the conjecture based on the Knowledge Base by applying one or more inference techniques. Figure 2.1 illustrates the high-level

architecture of a generic ATP system. The inference techniques included in the figure will be discussed shortly. In the figure, a concrete example of a conjecture that an ATP system would be able to solve reasonably quickly is included – proving the Inverse of Group Product from the Group Axioms.

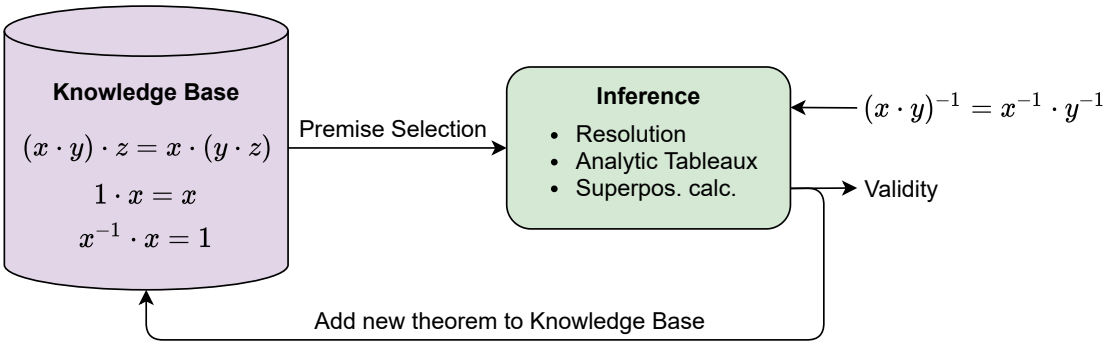


Figure 2.1: The high-level architecture of a generic ATP system.

It is important to note that "to prove something" is fairly loosely defined. More precise definitions of what is meant by a proof in a formal setting are (1) *satisfiability*: there exists some assignment of the variables (also known as a *model*) such that this assignment reduces the conjecture to logical *True*, given the Knowledge Base (Russell and Norvig, 2010, p. 250), and (2) *validity*: all assignments of variables reduces the conjecture to logical *True*, given the Knowledge Base (Russell and Norvig, 2010, p. 249).

The key component in an ATP system is the *inference* engine. ATP systems have large libraries of already proven facts, and it is the system's job to infer new facts automatically. Inference is, therefore, at the heart of all ATP systems. Most inference techniques use a so-called *proof by refutation* (Russell and Norvig, 2010, p. 250). Proof by refutation works by first negating the conjecture and showing that the negation of the conjecture is unsatisfiable (i.e., not satisfiable), which proves that the original conjecture is valid. This is true because a conjecture is valid if, and only if, its negation is unsatisfiable (Russell and Norvig, 2010, p. 250). The inference techniques involve a preprocessing step where the set of expressions is formulated using Conjunctive Normal Form (CNF) (Russell and Norvig, 2010, p. 345). In short, the set represents a conjunction (i.e., the logical AND $\wedge$) of clauses, where each clause is a disjunction (i.e., the logical OR $\vee$) of terms.

Another important aspect of ATP systems, is the *premise selection* step (Blanchette et al., 2016) (depicted in Figure 2.1). When performing inference, the system usually experiences an explosion in the combinatorial search space. Therefore, it is desirable only to include the background theory necessary to prove a conjecture and nothing more (Hoder and Voronkov, 2011). This task is known as premise selection.

Next, instead of going into details about specific ATP systems, the most common inference techniques are introduced in a general setting. This covers the theory needed to understand ATP systems while focusing on the important conceptual aspects and not on implementation details. Note that, while this section is restricted to the three most heavily adopted techniques, modern ATP systems usually deploy a combination of several different inference techniques. Other popular techniques, not covered here, include *generalized Modus Ponens*, *Model Elimination/Model Checking*, and the *Davis–Putnam–Logemann–Loveland algorithm* (Davis et al., 1962).

2.1.1 Resolution

The resolution technique (Russell and Norvig, 2010, p. 347 - 356) used in ATP systems combines variable substitution and the resolution inference rule. The idea is to resolve two disjunctions by unifying the variables in such a way that straightforward application of standard resolution is possible.

In order to illustrate the resolution inference technique, consider the following Knowledge Base (in CNF):

$$\text{Knowledge Base} = \{\neg P(x) \vee Q(x), \neg Q(y) \vee S(y), P(z)\},$$

with the goal to prove $S(A)$. The first step is to negate the conjecture: $\neg S(A)$. Then, inference by resolution will yield the resolution tree depicted in figure 2.2, and the conjecture is proven by refutation. This happens when resolution yields an empty set of clauses. Resolution is known to be refutation-complete, in that a set of clauses is unsatisfiable if and only if there exists a derivation of the empty clause using resolution alone (Russell and Norvig, 2010, p. 345).

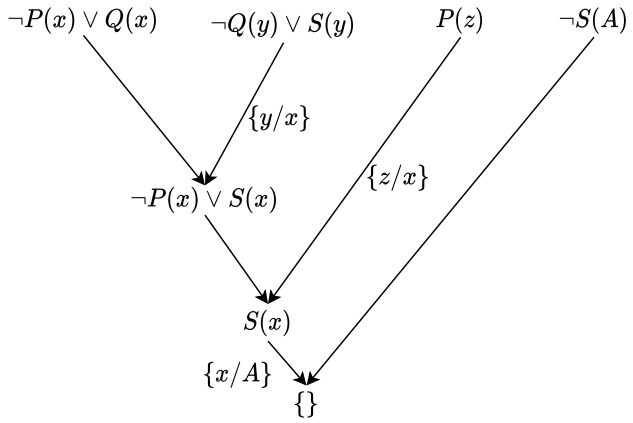


Figure 2.2: Example of a resolution tree.

2.1.2 Analytic Tableaux

Analytic Tableaux (Smullyan, 1968, p. 52-63) is a family of inference techniques, where the main idea is to break an expression into sub-expressions by a given set of rules for the logical connectives and quantifiers. This yields a tree structure (the *tableau*), where the leaves consist of atomic expressions that cannot be broken down further. A branch in the tree is considered *closed* when it inhabits a term and its negation. The original expression is unsatisfiable when all branches are closed, meaning the conjecture is proven by refutation.

In general, the tableau is expanded based on the following rules for logical connectives and quantifiers:

- $\wedge$: If a branch of the tree contains $A \wedge B$, add A and B to the leaves of that branch.
- $\vee$: If a branch of the tree contains $A \vee B$, split each leaf of the branch into two new leaves; one containing A and one containing B.
- $\neg$: If a branch of the tree contains $\neg(A \vee B)$ or $\neg(A \wedge B)$, use De Morgan's law to "push" the negation inwards.
- $\Rightarrow$ or $\Leftrightarrow$: If a branch of the tree contains $A \Rightarrow B$ or $A \Leftrightarrow B$, use the implication identity or the equivalence identity.
- $\exists$: Get rid of $\exists$ by existential instantiation.
- $\forall$: Get rid of $\forall$ by universal instantiation.

Many variations are possible. For example, one can delay branching as long as possible in order to avoid duplicate work, and do universal instantiation by the so-called *most general unifier* (Russell and Norvig, 2010, p. 327). Intuitively meaning that we want the instantiation variable to correspond to as many already instantiated variables as possible, so that a direct comparison of terms can be done.

Consider the example

$$\text{Knowledge Base} = \{\exists x.\neg P(x) \wedge \neg Q(x), \forall y.Q(y) \vee S(y)\}$$

where the goal is to prove $S(A)$. Using Analytic Tableaux, the tableau illustrated in Figure 2.3 is obtained. Each branch of the tree contains a contradiction (marked by dotted line), and $S(A)$ is therefore valid.

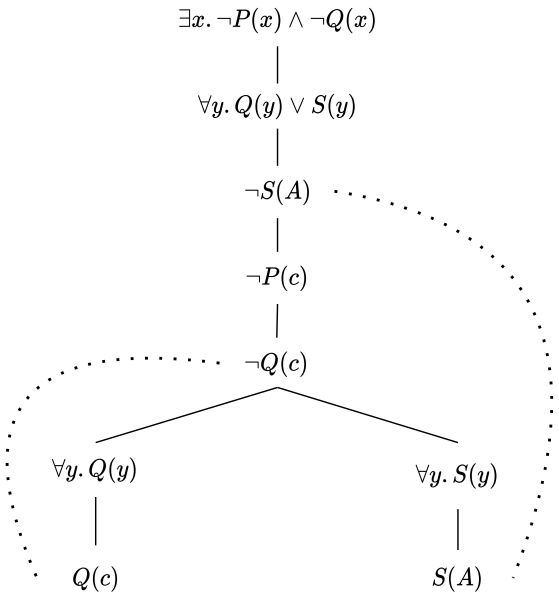


Figure 2.3: Example of a tableau. The dotted lines indicate conflicting terms (i.e., closed branches). All branches are closed in this tableau, meaning the original conjecture is valid by refutation.

2.1.3 Superposition Calculus

Most traditional ATP systems revolve around First-Order Logic *with* equality. The axiomatic way for dealing with equality (i.e. introducing new rules to the Knowledge Base that dictates how equality is handled) is usually inefficient, so designers have instead turned to another concept: *Superposition Calculus* (Rusinowitch, 1991; Schulz, 2002). Superposition Calculus involves the introduction of a new inference rule dictating how the system deals with equality:

$$\frac{C_1 \vee s = t, C_2 \vee P(s')}{\sigma(C_1 \vee C_2 \vee P(t)}), \text{ where } \sigma = \text{most general unifier of } (s, s')$$

This style of inference under equality creates a *rewrite system* where equations are subject to some ordering $\succ$ of terms. Ordering is a way to ensure termination, because it dictates in which "direction" to apply the rewrite associated with an equality (e.g., if we have $x = y$, should we set occurrences of x equal to y or set occurrences of y equal to x?).

A common problem in superposition calculus is the failure to achieve a *confluent* rewrite system. The rewrite system is considered confluent only when it deterministically outputs the expanded Knowledge Base, without considering the order of rewrite application (e.g., if we have $x = y$ and $a = b$, should be rewrite using $x = y$ first or rewrite using $a = b$ first?). The way to get around this problem is by applying

so-called *Knuth-Bendix completion* (Knuth and Bendix, 1970). The general approach is the following:

- Identify "critical pairs" (pairs of equations where confluence fails) by leveraging unification.
- Add critical pairs to Knowledge Base with the correct ordering. And repeat.

Superposition calculus is quite powerful. In the example from Figure 2.1, repeated mechanical application of the superposition calculus technique on the Knowledge Base results in:

$$\begin{aligned}\text{Knowledge Base} = \{&(x \cdot y) \cdot z = x \cdot (y \cdot z),\\ &1 \cdot x = x,\\ &x^{-1} \cdot x = 1,\\ &(x \cdot y)^{-1} = x^{-1} \cdot y^{-1},\\ &(x^{-1})^{-1} = x,\\ &1^{-1} = 1,\\ &x \cdot x^{-1} = 1,\\ &(x \cdot x^{-1}) \cdot y = y,\\ &x \cdot 1 = x,\\ &(x^{-1} \cdot x) \cdot y = y\}\end{aligned}$$

Notice that the conjecture from Figure 2.1 is now part of the Knowledge Base. That is, Inverse of Group Product simply drops out from the Group Axioms by a straightforward application of superposition calculus.

2.2 Interactive Theorem Proving

Traditional Interactive Theorem Proving (ITP) systems are not designed for theorem proving automation. Instead, they are used in cooperation with a human user. ITP research is a large subject, and diverse approaches and systems exist within this line of research. On one extreme, there are systems that act as safeguards and only formally verify proof made by a human. On the other extreme, systems can be "almost automated" and only subject to a small degree of human guidance during proof search. Harrison et al. (2014) provides an excellent introduction to the field and its history and Nawaz et al. (2019) a comprehensive comparison of different ITP systems.

Here, the scope is restricted to the concepts needed to understand Auto-ITP. These concepts are introduced in a general setting rather than via any concrete system. Some simplifications and generalizations are made as implementation details of specific systems are not central. However, some extra details are included on the ITP system Coq – the central ITP systems concerned in this Thesis. Other ITP systems used for Auto-ITP include HOL Light (Harrison, 1996) and HOL4 (Slind and Norrish, 2008).

2.2.1 Tactic-based Interaction

Most ITP systems implement so-called *tactics* (Harrison et al., 2014; Nawaz et al., 2019). This allows the user to interact with the system in a "backward searching" manner, meaning that the user starts with the goal (i.e., the conjecture) and breaks down this goal into simpler and simpler subgoals by applying tactics. This process continues until only trivially true (e.g., $1 = 1$) subgoals are left. Although tactic-based interaction is not the only option[1], it is this approach used in Auto-ITP. It is also normally the way human users interact with ITP systems (Harrison et al., 2014).

When interacting with an ITP system using tactics, a *search tree* is built (Bansal et al., 2019a). In this tree, the root is the original *top-level* goal, internal nodes consist of subgoals, and edges are associated with an applied tactic. Leaves are reached when a goal is trivially true. A node is *closed* when all its subgoals are proved. Figure 2.4

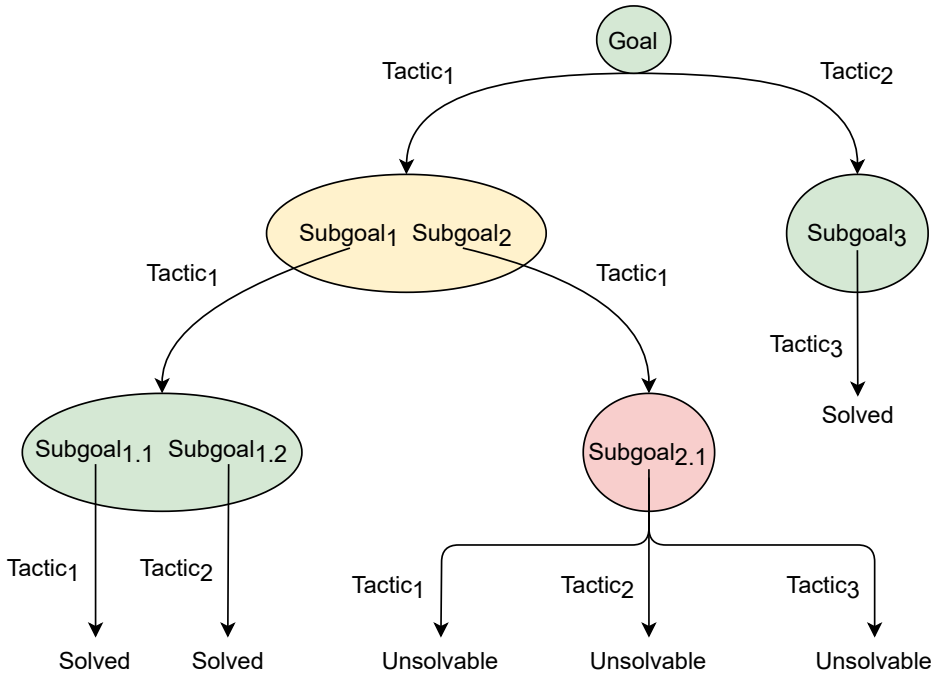


Figure 2.4: Example of a hypothetical proof tree. Green indicates closed nodes, yellow open nodes, and red nodes that are unprovable with the available tactics.

illustrates a hypothetical proof tree. In this example, three tactics are available: $Tactic_1$, $Tactic_2$ and $Tactic_3$. The user has first applied $Tactic_1$, resulting in the left-hand side subtree. The user has then decided to pursue this subtree further before encountering $Subgoal_{2.1}$. This subgoal is unsolvable with the three available tactics. The user has then

[1]One can, for example, interact in a "forward" fashion, using rules like *Conjugate* and *Modus Ponens* to build from a set of background knowledge (Harrison et al., 2014). This is more similar to how traditional ATP systems work.

backtracked to the root and applied $Tactic_2$, resulting in the right-hand side subtree. This branch is solved by applying $Tactic_3$ to the only subgoal, $Subgoal_3$, in the node. In this way, the top-level goal is solved too. A more seasoned user might have applied $Tactic_2$ and $Tactic_3$ right away, avoiding the left-hand side subtree altogether.

Tactic-based ITP systems usually support their own custom *tactic language* (Nawaz et al., 2019). Under the hood of such ITP systems, there is a parser that interprets the specifics of a tactic script (i.e., a sequence of tactics) and reduces the steps to code that is interpretable by the underlying programming language. Different systems support different tactics. In general though, they always represent high-level strategies in the proof procedure. A tactic can for example be an *induction* tactic or a *rewrite* tactic, corresponding to a proof by induction or a rewriting of the goal (usually using some already proven theorem), respectively.

2.2.2 Tactic Arguments and Proof Context

A user can sometimes pass *arguments* along with tactics (Barras et al., 1997; Harrison, 1996). Tactic arguments can generally be interpreted as part of one of two proof *contexts* (Yang and Deng, 2019; Bansal et al., 2019a). This Thesis will refer to these as the *global* context and the *local* context and interprets them in the following way (based on the the interpretations in Auto-ITP frameworks (Yang and Deng, 2019; Bansal et al., 2019a; Gauthier et al., 2020; Huang et al., 2019)).

The global context defines all background knowledge available at search time. This is analogous to an ATP system's Knowledge Base. That is, the global context contains already proven theorems, that can be used as tactic arguments in the proof procedure. For example, when applying a rewrite tactic, it can be useful to pass an equality theorem with the tactic, in order for the ITP system to know how to rewrite the current subgoal. This is essentially the same as the premise selection step done by ATP systems.

The local context includes the current goal/subgoal being evaluated, in addition to local *hypotheses*. Local hypotheses can occur as part of a tactic application. For example, when proving that $n+0=0$ using an induction tactic, the problem can be split in two: the base case $0+0=0$ and the inductive case "if for any $n=k$ we have $k+0=k$, then $k+1+0=k+1$". In this example, the ITP system will generate a new node containing two subgoals, each within an associated local context: (1) a subgoal for the base case $0+0=0$ where the local context does not include any hypotheses, and (2) a subgoal for the inductive case $k+1+0=k+1$ where the local context includes the hypothesis $k+0=k$.

Some generalizations are made in this Thesis, in order to make it easier to talk about the ITP proof procedure:

- **Core tactic**. The core tactic is a tactic without any arguments. Some core tactics

can be applied right away, while others require arguments to work. A core tactic will sometimes be denoted τ in this Thesis.

- **Tactic argument**. A tactic argument is either a theorem from the global context or a reference in the local context. The local context reference refers either to a term in the current subgoal or a local hypothesis. Most ITP systems support other types of arguments as well, but they are less typical and will not be considered in this Thesis. Theorems from the global context will sometimes be denoted $\boldsymbol{t}$ and hypotheses from the local context by $\boldsymbol{h}$.

- **Tactic application**. A tactic application consists of a core tactic and arguments from the local and global context. Arguments can be empty, depending on the core tactic. It will sometimes be denoted $\mathcal{T}$ in this Thesis.

Terms in the subgoal can generally be moved to the list of local hypotheses using specific tactics, resulting in a transformed but equivalent local context. This means that the local context argument can in practice be considered as only local hypotheses without restricting the space of tactic applications, if such transformations are applied.

2.2.3 Internal Automatic Engines

ITP systems often have internal small-scale automatic inference engines built-in (Hurd, 2003). These are invoked by tactic calls and often work by translating the goal and the tactic arguments into First-Order Logic before applying traditional First-Order inference techniques. This allows the user to solve subgoals that are not trivially true with a single tactic application. These engines are interesting in the context of Auto-ITP because they can serve as initial baselines (Yang et al., 2016) and are available for the Auto-ITP model to solve non-trivial subgoals.

2.2.4 The Logic of Computable Functions Principle

Modern ITP systems follow the so-called *Logic of Computable Functions* (LCF) principle (Geuvers, 2009). Two properties are emphasized in LCF-based ITP systems (Harrison, 2009):

1. The system revolves around a dedicated core: the *kernel*. The kernel consists of a set of inference rules, usually referred to as the *primitives*.

2. The system is implemented in a functional programming language. Type checking in the programming language ensures that all new inference rules in the system eventually reduce to the primitives.

The main idea is that if the kernel and the type checking mechanism in the programming language are sound, then the LCF approach is able to guarantee correctness of all new theorems entering the system. This is because all theorems that the system encounters will have to pass through the kernel and thus the system will know if the new theorem is

consistent with the primitives.

To implement this, the general-purpose functional programming language *Meta-Language* was developed (Gordon, 2000; Harrison et al., 2014). It works so that inference and theorems are of the same type in the language: `thm`. This makes it possible to implement `thm` as an abstract type of the primitives, which ensures *validity by construction* for new instances of type `thm` (Harrison et al., 2014).

The kernel also allows LCF-based systems to adhere to the *De Bruijn criterion* (Geuvers, 2009). This criterion states that any proof generated by the ITP system should be checkable by an (ideally simple) proof checker. Furthermore, this checker should be self-contained and independent of anything outside of itself. That is, the proof checker is a trustable and sound "black box" which can guarantee the correctness of proofs. This is precisely what the kernel in LCF-based systems provides.

2.2.5 Coq

The ITP system Coq (Barras et al., 1997) was released in 1989 and has been used in several formalization projects. Some of the most well known include formal proofs of the the Feit-Thompson theorem (Gonthier et al., 2013) and the Four Color theorem (Gonthier, 2008). A C compiler has also been formally verified using Coq (Leroy, 2016), in addition to the correctness of a Union-Find implementation (Conchon and Filliâtre, 2007). In other words, Coq is used for several problem types, not only pure mathematics. Each formalization project contains several formal Coq proofs. The large number of proofs is a primary reason why Coq is attractive as an underlying ITP system for Auto-ITP frameworks (Yang and Deng, 2019; Huang et al., 2019). Coq also has its own dedicated website[2], a standalone Integrated Development Environment, and is open source[3].

Coq implements a tactic language called *LTac* (Delahaye, 2000), defined by a context-free grammar (CFG). In short, grammar entries start with core tactics with production rules defining how to expand the tactic to include arguments. Some core tactics are terminal grammar entries, meaning that they do not take arguments. Others have the option of taking arguments, while some require arguments to work. For a full overview see the Coq reference manual (Barras et al., 1997), which is also available fully up-to-date in the form of a website: `https://coq.inria.fr/doc/`.

To keep things simple, this Thesis follows the Coq tactic space defined by the simplified CFG provided by Yang and Deng (2019) for the CoqGym framework (more on the CoqGym framework in Section 3.2.4). As explained by Yang and Deng (2019), statistics from Coq projects show that many tactics are rarely used in practice. The CFG

[2] https://coq.inria.fr/about-coq
[3] https://github.com/coq/coq

provided in CoqGym covers the most used Coq tactics. For details of the full grammar see Yang and Deng (2019). An overview of core tactics and their use of arguments is provided in Table 2.1. Furthermore, relevant tactics are summarized in the following list. Note that the tactics are interpreted using the CFG from Yang and Deng (2019), meaning their explanations are simplified compared to the full Coq documentation.

- `apply`. Matches the subgoal against global context arguments using First-Order unification. E.g., if the user knows that x implies y, and x is a known theorem in the global context, `apply` with x as an argument can be used to solve y. A local context argument can be used to specify if only a sub-expression should be matched, including local hypotheses.

- `rewrite`. Rewrites the subgoal using a global context argument. This only works if the subgoal and argument are equivalent. A local context argument can be used to specify if only a sub-expression should be rewritten, including local hypotheses.

- `intro/intros`. Puts universally quantified (i.e., "forall") variable in the list of local hypotheses. This can also be done for the left-hand side of implications. `intros` is the same as `intro` applied continuously until no more variables can be converted to a local hypothesis.

- `unfold`. If a term in the subgoal has a definition in the global context, `unfold` replaces the term by its definition. A local context argument can be used to specify if only a sub-expression should be unfolded, including local hypotheses.

- `induction`. Breaks up the subgoal into a base case and an inductive case by introducing an inductive hypothesis to the local context. A local context argument is used as an argument to identify the term to induct on. This can be a direct reference to a term in the subgoal or a reference to a local context hypothesis.

- `split`. Splits a subgoal consisting of a conjunction. For instance, splitting the subgoal $x \wedge y$ into two separate subgoals x and y.

- Main internal automatic engines: `trivial`, `auto`, `tauto`, `easy`, `intuition`, `ring`, `field`, `congruence`. These implement different strategies for automatic proofs of subgoals. E.g., `trivial` tries to apply a variety of other tactics under the hood and `auto` implements a full First-Order resolution procedure. Some are more specialized than others. E.g., `auto` is general-purpose while `ring` is specialized for subgoals consisting of addition and multiplication.

Table 2.1: Overview of Coq tactics. LC and GC refer to the local and global context, respectively. Only tactics from CoqGym's context-free grammar are included.

Core tactic	LC arg.	GC arg.	Core tactic	LC arg.	GC arg.
`intro`	No	No	`congruence`	No	No
`intros`	No	No	`left`	No	No
`apply`	Optional	Required	`right`	No	No
`auto`	No	Optional	`ring`	No	No
`rewrite`	Optional	Required	`symmetry`	No	No
`simpl`	Optional	No	`f_equal`	No	No
`unfold`	Optional	Required	`tauto`	No	No
`destruct`	No	Required	`revert`	Required	No
`induction`	Required	No	`specialize`	Required	Required
`elim`	No	Required	`idtac`	No	No
`split`	No	No	`hnf`	Optional	No
`assumption`	No	No	`inversion_clear`	Required	No
`trivial`	No	No	`contradiction`	Optional	No
`reflexivity`	No	No	`injection`	Required	No
`case`	No	Required	`exfalso`	No	No
`clear`	Optional	No	`cbv`	No	No
`subst`	Optional	No	`contradict`	Required	No
`generalize`	No	Required	`lia`	No	No
`exists`	Required	No	`field`	No	No
`red`	Optional	No	`easy`	No	No
`omega`	No	No	`cbn`	No	No
`discriminate`	Optional	No	`exact`	No	Required
`inversion`	Required	No	`intuition`	No	No
`constructor`	No	No	`eauto`	No	Optional

2.3 Machine Learning

Machine learning refers to a subfield of artificial intelligence concerned with methods that allow some *model* to learn over time. Typically, the machine learning model is a general-purpose *function approximator*, where a set of trainable parameters θ describe the function. The task of learning is the task of updating the parameters so that the model better approximates some ideal function. This Thesis is mainly interested in two subfields of machine learning: *supervised learning* and *reinforcement learning*.

Supervised learning (Russell and Norvig, 2010, p. 695) refers to machine learning methods that learn from a dataset of *labeled* examples. In this setting, one has a dataset of *feature vectors* where each feature vector $\boldsymbol{x}$ corresponds to some label y. A supervised learning model tries to learn the correlation between the datasets $X = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n)$ and $Y = (y_1, y_2, \ldots, y_n)$. When fed a new example, the

model makes a prediction $\hat{y}$. During training, $\hat{y}$ is compared to the true label y in order to compute some error term. The error term is used to incrementally correct the model's parameters so it more accurately predicts the true y for the next feature vector.

In reinforcement learning (Russell and Norvig, 2010, p. 830), the model does not learn from a set of labeled examples. Instead, the model interacts with some *environment*, which in turn leads to either positive or negative *reinforcements*. It is up to the model to learn what actions lead to positive reinforcements and what actions lead to negative. Typically, this means that the model will deploy some form of *trial-and-error* strategy, where a balance between *exploration* and *exploitation* is desirable (Russell and Norvig, 2010, p. 839). The reinforcement learning method relevant in this Thesis is *deep Q-learning* (Mnih et al., 2015).

Deep Q-learning is similar to supervised learning in that the model also takes in a feature vector $\boldsymbol{x}$ and makes a prediction $\hat{y}$. $\hat{y}$ will in this case be an action the model can do in the current state s_t. However, there is no true label y to compare $\hat{y}$ against. Instead, a *replay memory* is used to train the model. In this setup, there is an expected reward $\hat{r}$ whenever s_{t+1} (the state reached by applying action y in s_t) is not a terminal state and a true reward r when s_{t+1} is a terminal state. r (or $\hat{r}$) is compared to the models expected reward for applying y in s_t (known as the *temporal difference* (Russell and Norvig, 2010, p. 836)), which is used to correct the models perception of whether or not y was a good action in s_t. When a model trains using a temporal difference-based replay memory, it is trained in a self-supervised manner. This essentially means the model trains in a supervised way, where labels are generated by interacting with the environment.

The following subsections will cover relevant theory in machine learning. Because deep Q-learning utilizes self-learning, the concepts are explained primarily from a supervised learning point-of-view. Topics specific to deep Q-learning is explained in Subsection 2.3.13.

2.3.1 Features

Machine learning models take feature vectors as input. Each entry in this vector is called a *feature*. Simply put, a feature corresponds to some attribute from the domain at hand. Models need $\boldsymbol{x}$ to be in a format that is computer understandable. This means that the attributes have to be converted to some real-valued representation – an *encoding* of the attributes – in which $\boldsymbol{x}$ is a real-valued vector that can be used to train the model.

To understand what constitutes features, an Auto-ITP example will be used. Say the goal g is to prove the expression

$$a + 0 = a.$$

For simplicity, say also that there is a finite set defining all syntactical elements used to make expressions: $S = \{a, b, 0, =, -, +\}$. Then, a simple way to obtain a feature

representation of g is to *one-hot encode* the expression:

$$\boldsymbol{x} = (1, 0, 1, 1, 0, 1)$$

This is a representation where 1 on index i indicates that element i from S is present in the expression, and vice versa for 0.

A one-hot encoding, like the one described here, is easy to implement and is a common starting point for dealing with non-numerical *categorical* attributes. Categorical attributes are attributes that are (as might be clear from the name) discrete and belong to some category. However, $\boldsymbol{x}$ does not capture semantic information well. In addition, it suffers from the *curse of dimensional*, meaning that feature vectors can become extensively large when the set of all symbols is large. To deal with this, the encoding can be mapped to an *embedding* before a predictive model uses the embedding to make a prediction. A good embedding will capture semantic information and deal with the curse of dimensionality.

Other attributes might not need a one-hot encoding as they have a natural real-valued representation (e.g., the age or height of a person). Alternatively, it could be the case that no obvious real-valued representation exists for the attribute, and a one-hot encoding is also not a reasonable approach. In these cases, a more sophisticated encoding is needed. For example, Transformer models map sequence elements to set of *tokens* to obtain a feature representation (explained further in Subsection 2.3.12).

2.3.2 Classification Problems

A classification problem is a type of machine learning problem where each example belongs to a single *class*. In *binary* classification problems, there are two classes. An example of this is to predict whether a cat is in an image or not. The two classes would, in this case, be "no cat" and "cat".

All models in this Thesis are classification models. More specifically, they are *multi-class* classification models. Multi-class means that there are more than two classes to consider. The output of such a model is a probability distribution over the classes, where the class with the highest probability is the model's prediction.

Note that this Thesis does not consider *multi-label* multi-class models. That is, there is never more than one correct class for each example.

The Softmax Function

The Softmax function (Russell and Norvig, 2010, p. 848) is a function mapping any real-valued vector to a probability distribution of the same dimension of the input vector. This probability distribution is mapped so that the largest value in the input vector has the highest probability, the second largest has the second-highest probability, and so

on. Softmax is typically used as a final function in a multi-class classification model to achieve a probability distribution over the classes. It is given by

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{K} e^{y_j}},$$

where K is the number of classes and y_i denotes the ith class.

2.3.3 Mini-Batch Training

An important concept in training machine learning models is the idea of *mini-batch training* (Ruder, 2016). Mini-batch training refers to a method of training where models predict a mini-batch of the training data at a time. Training only occurs between each mini-batch, where the error term is computed over the whole mini-batch rather than on individual examples.

The size of the mini-batch can vary and impacts both performance and the time it takes to train a model (Smith et al., 2018). If available hardware resources can support it, increasing the mini-batch size will typically decrease the training time. This is because more (or all) of the examples in the mini-batch can be computed in parallel. When setting the mini-batch size to one, training simply follows an example-by-example routine. When setting the mini-batch size equal to the size of the entire training set, a batch-style training routine is followed (Ruder, 2016). If an example-by-example, mini-batch-style or batch-style training leads to the best performance usually varies from problem to problem.

2.3.4 Loss Function

For machine learning models to be able to learn, they need some feedback that indicates whether or not they are doing well or not well. This is usually dealt with by using a *loss function* (Russell and Norvig, 2010, p. 710). The loss function takes the prediction $\hat{y}$ and the label y as input and calculates the loss based on how "wrong" $\hat{y}$ was compared to y. Importantly, this loss is a real value which can be used to update the model's parameters. When deploying mini-batch training, the loss is usually the mean of the loss for each example in the mini-batch.

Many variations of loss functions are possible. Two loss functions are used in this Thesis: *Cross-entropy loss* and *Huber loss*.

Cross-Entropy Loss

Cross-entropy is defined for two probability distributions. To explain cross-entropy, assume the binary case with distributions $P = (p, 1-p)$ and $Q = (q, 1-q)$. The cross-entropy of Q relative to P is defined as

$$\text{cross-entropy, binary} = -(p\log(q) + (1-p)\log(1-q))$$

This is easily extended to the general case in which the probability distributions are over n elements:

$$\text{cross-entropy} = -\sum_{i=0}^{n} p_i \log(q_i)$$

Cross-entropy is used as a loss function by simply applying this formula directly, where the loss is calculated as the cross-entropy of $\hat{y}$ relative to y.

Cross-entropy loss is zero whenever $\hat{y} = y$. It grows relatively slowly for correct predictions (i.e., prediction > 0.5 when the label is one) and relatively fast for predictions that are wrong and where the model is fairly confident in its prediction (i.e., prediction $<< 0.5$ when the label is one). The idea is that the penalty is much stricter when the model is radically wrong and milder, but still not zero, when the model is correct but not confident.

Huber Loss

Huber loss, named after its inventor Peter J. Huber (1964), is a combination of mean-squared-error (MSE) and mean-absolute-error (MAE). MSE and MAE are defined as follows (for the binary case, where mini-batch size is set to one):

$$\text{MSE} = \frac{1}{2}(y - \hat{y})^2$$
$$\text{MAE} = \frac{1}{2}|y - \hat{y}|$$

The idea of Huber loss is to use MSE whenever the $|y - \hat{y}|$ is below a certain threshold, and MAE otherwise. In this way, the loss puts less emphasis on large losses and is, therefore, less sensitive to outliers. This is useful if the training process is unstable, which is typically the case training a deep reinforcement learning model like a deep Q-learning model (Mnih et al., 2015).

2.3.5 Evaluation

The loss function is the metric that guides the learning process for machine learning models. However, the loss function alone is rarely the metric used to evaluate models. For classification models, it is more typical to care about the *accuracy* of the model. The accuracy is the percentage of correctly labeled examples from a given set of examples.

It is common to split the dataset into two parts; one part being the *training* set and the other being the *test* set (also called the *holdout* set). It is important to "hold out" the test set from the model so that a fair, final evaluation can be performed. Furthermore, it is common to use some of the examples in the train set as a *validation* set. The model is tested on the validation set at even intervals during training. This is useful because it indicates how well the model is doing while training and hyperparameters can be modified dynamically while training.

2.3.6 Neural Networks

Neural networks (Russell and Norvig, 2010, p. 727 - 736) are machine learning methods inspired by the low-level physical structure of the brain. They are part of a family of methods called deep learning. Neural networks are built as networks of nodes and edges, where information passes along edges and through nodes. Each edge in the network has an associated *weight*. A node takes in the sum of the weighted values of all its input edges and computes an output value based on an *activation function* (discussed further in Subsection 2.3.9). Weights are, therefore, the trainable parameters of the neural network models. A neural network architecture is made up of *layers*, with an input layer, an output layer, and layers in between called *hidden* layers. In addition to weights, neural networks have something called *bias*. Bias is similar to constants in a linear function, whereby the function is shifted by the constant value. Bias is typically trained, just like the weights.

To train Neural Networks a method known as *backpropagation* is used (Russell and Norvig, 2010, p. 733). The general idea is to propagate the error through the network, from the output layer to the input layer. The gradients of the weights in each layer are efficiently computed based on the gradients in the prior layer, using the chain rule. In this way, gradients are calculated while avoiding redundant calculations and *gradient decent* can be applied layer-by-layer. This means that weights can be adjusted to minimize the loss by following the slopes of the loss function. In gradient descent, a step size is defined, dictating how much the weights should be adjusted in each iteration. This step size is commonly called the *learning rate* and denoted α. Neural networks are usually trained for several *epochs*. One epoch corresponds to one pass over the training data. Furthermore, validation is typically performed between each epoch.

Feed Forward Networks

A common type of neural metwork used for prediction is the Feed Forward Network (FFN) (Russell and Norvig, 2010, p. 729). The input layer takes in the feature vector, values are passed along edges and nodes, being manipulated by the activation functions, and the output layer outputs the prediction. Edges in an FFN point "forward" in the network. Figure 2.5 illustrates an FFN taking in the goal expression from Subsection 2.3.1 and outputting a probability distribution over two tactics.

2.3.7 Optimizers

When training neural networks, an *optimizer* (Ruder, 2016) is used. Optimizers dictate exactly how the propagation and gradient descent algorithm is implemented. Most optimizer leverage so-called *momentum*. Momentum is a way to favor the previous direction of the gradient descent by adding the previous update times a constant to the current update function. This creates the effect of "momentum" being kept from time step to time step in the search along the loss function's slopes. Momentum speeds up the gradient descent process and avoids oscillation.

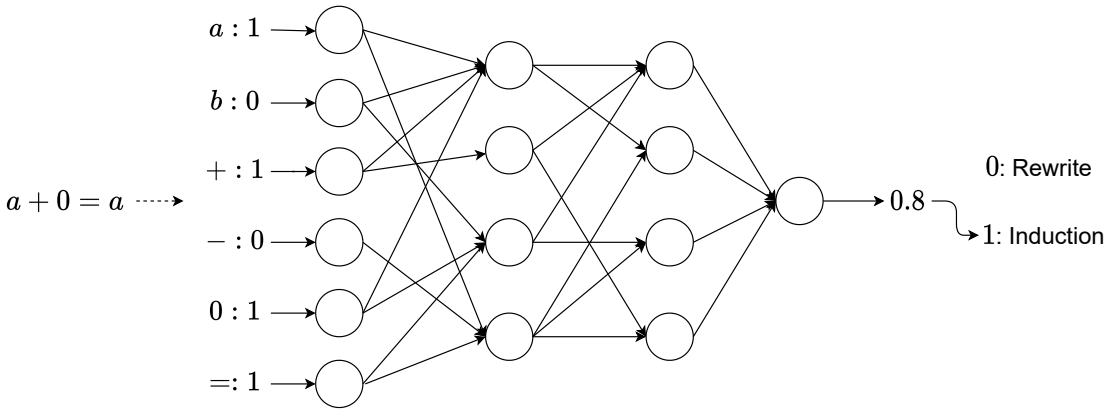


Figure 2.5: Example of a Feed Forward Network.

Adam

The Adaptive Moment Estimation (Adam) (Kingma and Ba, 2017) optimizer has become one of the most popular optimizers for deep learning (Ruder, 2016). It is (as the name suggests) an adaptive optimizer. This means that the optimizer adapts the learning rate based on some rules. In the case of Adam, the learning rate for each weight is adapted based on *momentum*.

Specifically, Adam uses an exponentially decaying average of previous gradients as part of the current update function. Furthermore, Adam uses two types of momentum: first-order m_t and second-order v_t momentum. The second-order momentum is past gradients squared. Also, accounting for the fact that first-order and second-order exponentially decaying momentum is biased towards 0, Kingma and Ba (2017) arrive at the following bias-correct momentum updates:

$$m_t = \frac{\beta_1 m_{t-1} + (1-\beta_1) g_t}{1-\beta_1^t}$$
$$v_t = \frac{\beta_2 v_{t-1} + (1-\beta_2) g_t^2}{1-\beta_2^t}$$

β is the decay rate and g_t is the gradient at time step t. The update rule for the model parameters is:

$$\theta_{t+1} = \theta_t - \alpha \frac{m_t}{\sqrt{v_t} + \epsilon}$$

ϵ is a small constant included to avoid division by zero.

2.3.8 Regularization

Regularization is an essential concept in machine learning. Regularization methods are used to combat *overfitting* of the training data. Overfitting is a phenomenon where the machine learning model is able to perform well on the training data but does not perform well on the validation or test data. The model essentially finds a correlation between X_{train} and Y_{train} that is too specific for the training set. This leads to the model not being able to generalize to other examples outside of the training set[4].

Regularization techniques generally try to penalize more complex models, in favor of simpler ones. Many techniques exist. Here, two are explained as they are the ones used by models in this Thesis: *weight decay* and *dropout*.

Weight Decay

Weight decay is a simple technique well known to combat overfitting (Krogh and Hertz, 1992). The basic idea is to decrease the complexity of the network by limiting the growth of weights. This is achieved by penalizing large weights using a cost term in the loss function[5]:

$$\hat{\text{loss}}(\theta) = \text{loss}(\theta) + \frac{1}{2}\lambda \sum \theta^2$$

In this way, smaller weights will be favored over large weights, decreasing the complexity of the network and combating overfitting.

Dropout

Dropout (Srivastava et al., 2014) is a technique where at each time step during training, each neuron in the neural network will have its output (its "contribution") set to zero. This happens with probability p (the dropout rate). Dropout works because it limits co-dependency between neurons, meaning that neurons become less dependent on the output of other neurons. This is a way to create a more "robust" network, which is less likely to overfit.

2.3.9 Activation Functions

Activation functions are often applied to the output of neurons to allow the network to approximate non-linear functions. This is because activation functions are non-linear mappings. Two activation functions are used in this Thesis: $ReLU$ and $Tanh$.

[4] The opposite phenomenon is called *underfitting*, where the model is not even able to find good predictions for the training set.

[5] This is very similar to so-called *L2 regularization*, and weight decay is therefore sometimes referred to as L2 regularization for neural networks.

ReLU

The Rectified Linear Unit (ReLU) function is given by

$$\text{ReLU}(x) = \max\{x, 0\}$$

It is a straightforward function and has become the most popular activation function (Nwankpa et al., 2021). It is faster than most other activation functions and shows strong generalization ability for deep learning models (Nwankpa et al., 2021).

Tanh

Hyperbolic tangent (tanh) maps inputs to the interval $(-1, 1)$. This allows it to keep contributions from negative outputs (something that ReLU does not), while at the same time making sure that no outputs grow too large (in either negative or positive direction). The function is given by:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

It is a common activation function for natural language tasks (Nwankpa et al., 2021).

2.3.10 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a special type of neural network designed to capture spatial relationships in images (Nielsen, 2015). Three key ideas are introduced in CNNs: *local receptive fields*, *shared weights* and *pooling*.

Instead of having the input neurons fully connected to every neuron in the first hidden layer, a CNN connects a cluster of spatially close neurons to the same hidden neuron. For the example of an image, the input neurons can be considered as a matrix corresponding to the pixels in the image. Each region in the pixel matrix is connected to a single neuron in the first hidden layer. The regions are defined by a *kernel* of fixed dimensions that "slides" across the pixel matrix, mapping local receptive fields to hidden units. This is known as the *convolutional* layer, also sometimes also called a *feature map* (Nielsen, 2015). Another key property of CNNs is that the weights of the hidden units in the convolution layer share the same weights and bias (Nielsen, 2015).

CNNs also have *pooling* layers (Nielsen, 2015). Pooling is the process of simplifying the feature map by mapping regions from the hidden layer to a new layer by some (simple) mathematical operation. For instance, 2x2 *max* pooling maps 2x2 regions in the hidden layer to the largest value contained in the 2x2 window. This is a parameter-free operation, meaning that the pooling layer does not contain weights and is therefore not trained during backpropagation.

2.3.11 Graph Neural Networks

Graph Neural Networks (GNNs) refer to deep learning methods applied to graph structures. This is achieved by so-called *message passing* techniques (Paliwal et al., 2020). In a typical setup, node embeddings are computed using message passing before predictions can be made on either individual nodes or the graph as a whole (Zhang et al., 2018). The message passing function takes in the embedding of a node v_i $\boldsymbol{x_i}$ and embeddings of nodes in the local neighborhood of v_i. For GNNs, the message passing function is a neural network and therefore consists of trainable parameters θ (i.e., the network weights). A typically embedding process involves several rounds of message passing, called *hops*. For K hops, a node embedding will depend on neighbors as far as K edges away. Self-loops are usually added, meaning that information from v_i's own embedding is not lost in the message passing process. Note, in the following subsection X does not denote a training set but rather the initial node embedding of the graph.

Three GNN methods are used in this Thesis: Graph Convolutional Networks (GCN) (Kipf and Welling, 2017), Simple Graph Convolutional Networks (SGC) (Wu et al., 2019) and Deep Graph Convolutional Neural Networks for end-to-end graph classification (DGCNN) (Zhang et al., 2018). GCN and SGC were proposed as node classification methods. However, this Thesis is concerned with graph classification, in which the graph itself is classified, and not individual nodes. GCN and SGC (and most types of GNN methods) can serve this purpose too by having some form of *readout* function. The readout function takes in the embedded nodes and maps them to a fixed-sized graph representation. The graph representation can, in turn, be used in a standard classification model. DGCNN is an architecture that does precisely this. It leverages the GCN technique for node embeddings and uses a novel sorting-based readout function. Each of the three methods will now be covered in more detail.

Graph Convolutional Network

The starting point for the Graph Convolutional Networks (GCNs) describe by Kipf and Welling (2017) is a semi-supervised node classification problem. This means that labels are available for a subset of nodes, but not all. The problem is then to predict the labels for the remaining nodes.

In short, Kipf and Welling (2017) solve this problem by considering the labeled nodes as training data for a neural network model, in which both the node embedding matrix X and the adjacency matrix A are inputs. In this way, both node embeddings and the relationships between nodes is part of the feature space. This means that crucial relational semantics encoded by the graph structure is included in the message passing process.

A K-layer GCN is identical to propagating node feature vector through a K-layer FFN, with the addition that the hidden representation of each feature $\boldsymbol{w}_i$ is averaged with the feature vectors of its local neighborhood (Wu et al., 2019). This is

analogous to a convolution in CNNs (hence the name Graph *Convolutional* Network).

This achieved in the following way. GCN first adds self-loops to A, using the identity matrix I: $\tilde{A} = A + I$. Then, $\tilde{A}$ is normalized using its diagonal degree matrix $\tilde{D}$:

$$S = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$$

The convolutional step in GCN for the kth hidden representation $W^{(k)}$ can then be compactly described by:

$$\bar{W}^{(k)} \leftarrow SW^{(k-1)}$$

This is known as *feature propagation* (Wu et al., 2019) and is similar to feature mapping in CNNs. The next step is to apply linear transformation by passing $\bar{W}^{(k)}$ through a parameterized function; an FFN with trainable parameters θ, before nonlinear activation is applied:

$$W^{(k)} \rightarrow \sigma(\bar{W}^{(k)}\theta)$$

The overall algorithm is depicted in Figure 2.6. The figure showcases how GCN implements a 3-step process: feature propagation, linear transformation, and nonlinear activation. ReLU is used as the activation function σ in the figure. Softmax is applied at the end to obtain a classification.

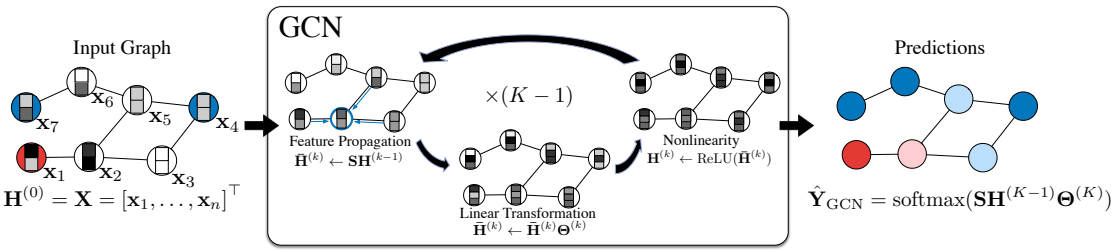


Figure 2.6: Illustration of the GCN message passing algorithm. Figure from Wu et al. (2019), with permission from Felix Wu.

Simplified Graph Convolutions

The Simplified Graph Convolution (SGC) message passing technique (Wu et al., 2019) makes two key simplifications to the GCN technique. The first is to remove the nonlinear activation function between each GCN layer. Wu et al. (2019) hypothesize that activation between messages is not crucial for capturing relational semantics. The resulting classifier becomes:

$$\mathrm{SGC} = \mathrm{softmax}(SS \ldots SX\theta^{(1)}\theta^{(2)} \ldots \theta^{(K)})$$

for K hop message passing. The notation is simplified further by collapsing the normalized adjacency matrix multiplications into a single operations where S is raised to the power of K. The weights can also be reparameterized into a single matrix θ:

$$\text{SGC} = \text{softmax}(S^K X\theta)$$

SGC is depicted in Figure 2.7. As noted by Wu et al. (2019), SGC is easy to interpret. It first consists of a parameter-free feature extraction step (the message passing) $\bar{X} = S^K X$, before a classification step outputs the prediction (shown as "Logistic Regression" in Figure 2.7) $\hat{Y} = \text{softmax}(\bar{X}\theta)$.

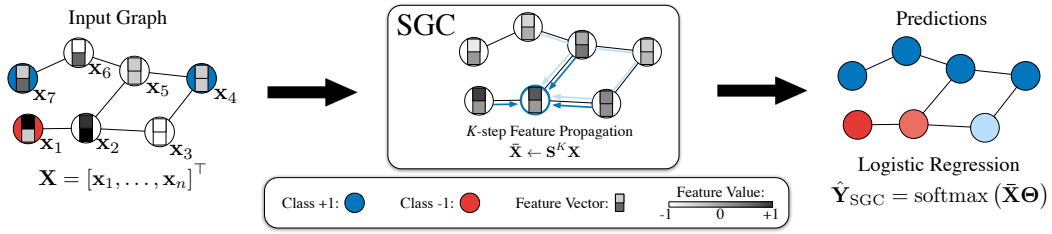


Figure 2.7: Illustration of the SGC message passing algorithm. Figure from Wu et al. (2019), with permission from Felix Wu.

Deep Graph Convolutional Neural Network

Deep Graph Convolutional Neural Network (DGCNN) (Zhang et al., 2018) is an end-to-end architecture for graph classification. The overall architecture is depicted in Figure 2.8. The first step is K rounds of GCN message passing. For each round, a node embedding W is stored. Zhang et al. (2018) propose the following GCN implementation:

$$W^{(k+1)} = \sigma(\tilde{D}^{-1}\tilde{A}W^{(k)}\theta^{(k)})$$

As with GCN from Kipf and Welling (2017), $\tilde{A}$ denotes the graph adjacency matrix with added self-loops $\tilde{A} = A + I$, and $\tilde{D}$ is the diagonal degree matrix of $\tilde{A}$. In other words, this implementation is almost identical to the GCN implementation from Kipf and Welling (2017). Node embeddings from each round of message passing is concatenated into a single graph representation $W^{(1:K)} = [W^{(1)}, \ldots, W^{(K)}]$, with $W^{(0)} = X$.

The next step is to perform a readout of the node embeddings. DGCNN achieves this by a novel *SortPool* layer, which extracts the top n rows from $W^{(1:K)}$. This is done by first sorting $W^{(1:K)}$ based on the final message passing computation $W^{(K)}$. Zhang et al. (2018) show that the output from the message passing rounds can be viewed as continuous Weisfeiler-Lehman (WL) node colors (Weisfeiler and Lehmann, 1968). In short, WL colors are node colors obtained by iteratively updating colors based on the node's previous color and the color of its local neighborhood[6]. The final output $W^{(K)}$ is

[6] "Color" is not meant to be interpreted literally, but rather as a *fingerprint* representing the node.

the most "refined" such coloring, and therefore the basis for the sort (Zhang et al., 2018). Crucially, SortPool provides a consistent graph representation. This means that if two graphs are isomorphic[7], their graph representation after SortPool is the same Zhang et al. (2018).

SortPool can pass gradient loss back to the GCN layers. Learning end-to-end graph classification is therefore possible with DGCNN. Zhang et al. (2018) also pad the output from SortPool always contains exactly n rows. Finally, a traditional CNN network is implemented as a prediction network.

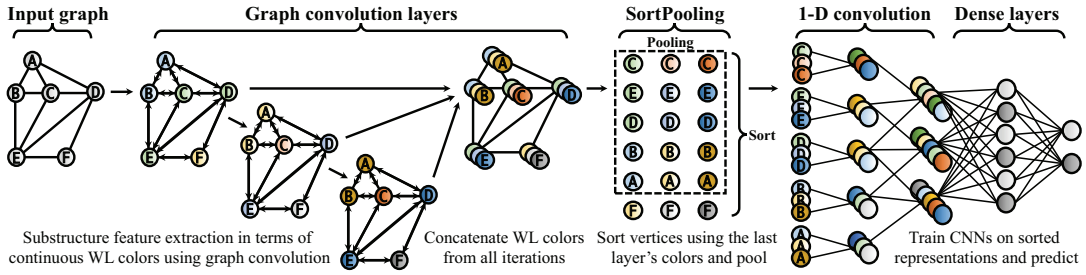


Figure 2.8: Illustration of the DGCNN end-to-end graph classification architecture. Figure from Zhang et al. (2018), with permission from Yixin Chen.

2.3.12 Transformers

The Transformer (Vaswani et al., 2017) is a deep learning architecture developed in the field of Natural Language Processing (NLP). The architecture implements two modules: an *encode* and a *decoder*. In addition, K encoders and decoders are stacked. The input to these modules is tokenized representations of natural language, and the output is a probability distribution. The key components in the Transformer are *multi-headed self-attention* networks, responsible for capturing semantic information in the input sequence. Transformers are so-called *autoregressive* systems, meaning that previous outputs are part of the current input. The main components will now be explained in more detail.

Tokens and Input Embedding

The tokenization step maps an input sequence to a vector of tokens. For instance, the input "My name is Bob" might be mapped to the tokens [My, name, is, Bob]. Before the Transformer encoder and decoder can make computations on the tokens, each token is mapped to a real-valued vector representation. For example, [My, name, is, Bob] might be mapped to $[0.5, 0.3, 0.2, 0.8]$. The input embedding can be obtained in several ways. Often a pre-defined map is used to ensure that the same word always has the same initial

[7]Isomorphism between two graphs G_1, G_2 means that their exists a one-to-one mapping between nodes in G_1 and G_2.

embedding vector. Embedding vectors can be based on pre-trained models that have learned a meaningful mapping.

Positional Encoding

Before input embeddings are fed into the encoder and decoder, *positional encodings* are computed for each input embedding. A positional encoding contains additional information about the absolute and relative position of tokens in a sequence. Vaswani et al. (2017) compute positional encoding in the following way:

$$PE(pos, 2i) = sin(pos/10000^{2i/d_{model}})$$
$$PE(pos, 2i+1) = cos(pos/10000^{2i/d_{model}})$$

pos is the position of the token in the sequence, i is the dimension of the positional encoding, and d_{model} is the input embedding dimension. These functions are chosen because PE_{pos+k}, where k is an offset, is a linear function of P_{pos}. This means that the relative position of a token is well formulated in the positional encoding, in addition to the sinusoid representation of absolute position.

Multi-Head Self-Attention

Both the encoder and the decoder consist of multi-head self-attention networks. Multi-head refers to the fact that several self-attention layers are stacked and run in parallel before the output from each is concatenated and run through a final linear layer.

Self-attention is a technique able to focus its "attention" on the most important tokens in the input embedding, based on the relationship between the tokens. The self-attention layer takes in a query Q and key-value pairs K, V (in matrix form). Q is the current word, and the key-value pairs represent the "memory" of all the words that have been generated up to that point. Vaswani et al. (2017) call their attention *Scaled Dot-Product Attention*, because attention is computed based on the dot-product between the input matrices and scaled based on the dimension of the key matrix d_k:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Encoder and Decoder

The encoder takes in the input embedding and the positional encoding. The is transformed into query Q and key-value pairs K, V and fed into a multi-headed attention layer. Both the input and the multi-head attention are added together and normalized in a new layer. Finally, an FFN is used to obtain a linear combination before the FNN input and output is again added and normalized to obtain a final encoder output.

The decoder is similar to the encoder. However, an additional multi-head attention layer is used over the output of the encoder and the output of the first multi-head attention layer in the decoder. The decoder is autoregressive, meaning it generates tokens one at a time while being fed in the previous outputs. The input is also right shifted one position to ensure that the prediction at position i only depends on the known outputs at positions before i.

BERT

Bidirectional Encoder Representations from Transformers (BERT) (Devlin et al., 2018) is a Transformer-based architecture allowing embeddings to be computed based on both the left and right (i.e., *bidirectional*) context of the token. A key idea is that BERT can first be pre-trained on general tasks (so-called *upstream* training) and later fine-tuned to a specific task (so-called *downstream* training). BERT has to attend to both preceding and succeeding tokens during pre-training.

BERT consists of several layers of fully connected Transformers. Two versions are implemented by Devlin et al. (2018). BERT-base consists of 12 transformer blocks, each made up of 12 attention heads. Hidden representations are of size 768, making BERT-base consist of 110 million parameters. A BERT-large version is also implemented, which implements more transformer blocks and attention heads. BERT-base is the only relevant BERT implementation for this Thesis.

BERT uses a so-called *WordPiece* tokenization technique. In short, an input embedding is extracted from natural language by breaking words into tokens from a set of more than 30,000 tokens. It is possible to input more than one sentence to BERT by separating sentences using a special separation token.

Devlin et al. (2018) define two pre-training tasks for BERT. One is to mask tokens with a 15% probability. BERT is then tasked with predicting the masked tokens. The second is a next sentence prediction task. Given sentence A and B, BERT has to predict whether or not B is the next sentence after A in the dataset. The total number of words in the pre-training data is 3.3 billion.

2.3.13 Deep Q-Learning

Deep Q-learning (Mnih et al., 2015) is a reinforcement learning technique, combining Q-learning and deep learning. Here, Q-learning will be explained first, before the deep learning aspect of deep Q-learning is explained. The important exploration-exploitation trade-off will also be explained.

Q-Learning

Q-learning (Russell and Norvig, 2010, p. 831) is a so-called *model free* reinforcement learning method. A model in reinforcement learning refers to an explicit transition probability distribution over the possible state-action pairs in the environment and a reward function mapping states to reinforcements. Q-learning works without any such explicit model.

In Q-learning, a function $Q(s, a)$ is learned through trial-and-error. $Q(s, a)$ maps a state-action pair (s, a) to some real-value. This value represents the expected utility obtained by performing action a in state s. If a good $Q(s, a)$ is learned, an effective Q-learning agent can operate by choosing to perform the best action a^* for each state:

$$a^* = \underset{a}{\operatorname{argmax}}\, Q(s, a)$$

The Q function is trained by using the Bellman equation (Russell and Norvig, 2010, p. 652), where the *temporal difference* δ (Russell and Norvig, 2010, p. 836) between state s_t and s_{t+1} is the critical component. The Bellman equation is the following *value iteration* update function:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta,$$

where

$$\delta = r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)$$

r_t is the reward obtained when reaching s_{t+1}. r_t is often a neutral reward whenever s_{t+1} is not a terminal state, and either positive or negative reward whenever s_{t+1} is a desirable or not desirable terminal state. γ is the *discount factor*. The purpose of γ is to discount the importance of estimated future rewards, incentivizing closer rewards over distant rewards. The temporal difference is weighted by a learning rate α.

$\max_a Q(s_{t+1}, a)$ means that the model uses the (so-far) best known policy to estimate the future reward obtainable from s_{t+1}, called *off-policy* learning. Off-policy agents learn the value of the optimal action policy, independently of the agent's actions[8].

Q-learning works because the estimate $Q(s_t, a_t)$ is gradually refined based on the true outcome from performing a_t in s_t. The temporal difference essentially encodes the error between the estimate at time step t and the *slightly better* estimate at time step $t + 1$. Importantly, a reward at a terminal state reveals the ground truth about the final state, which makes learning possible.

[8]The opposite is known as *on-policy* learning. In this setup, the agent will learn the value of the true policy being carried out by the agent, which typically includes non-optimal explorative actions.

Approximating $Q(s, a)$ via Neural Networks

Traditional Q-learning encodes the Q function as an explicit dictionary-style map. However, this is not feasible in complex environments where the space of states (and possibly actions) is large. A way to overcome this is approximating for $Q(s, a)$ via a learnable function approximator. This is the main idea of deep Q-learning (Mnih et al., 2015). Neural networks are universal function approximators and can be used as an efficient alternative to explicit Q maps.

A *replay memory* is used to train deep Q-learning agents (Mnih et al., 2015). Whenever the agent performs a new action a_t the tuple $Q(s_t, a_t), Q(s_{t+1}, a^*_{t+1}), r_t)$ is added to the replay memory. This is a new *experience* that the agent can learn from. Experiences in the replay memory are used to train the model in a self-supervised manner, where the labels are are $r_t + Q(s_{t+1}, a^*_{t+1})$ and the prediction is $Q(s_t, a_t)$. The error then becomes the temporal difference and can be used for backpropagation training. Learning from a replay memory, therefore, closely resembles supervised learning. The difference is that the labels are imperfect estimates of true values that gradually improve as the true terminal rewards drive the estimates towards more and more correct estimates.

In practice, a few enhancements are used to improve the replay memory technique. Instead of training on all experiences in the replay memory, only a subset is used. These are typically chosen uniformly at random. This has shown to result in a more stable learning process (Mnih et al., 2015). Also, a separate network Q_{target} is used to calculate $Q(s_{t+1}, a^*_{t+1})$ (Mnih et al., 2015). This network is known as the *target* network. The target network is periodically updated with the Q-network's weights, making it converge towards better and better estimates while always "lagging" behind the Q-network. This has shown to result in a more stable learning process (Mnih et al., 2015). In practice the Q-network takes in the current state and outputs a probability distribution over the entire action space, rather than taking in each state-action pairs one at a time.

Exploration vs. Exploitation

A key aspect of Q-learning (and reinforcement learning in general) is the idea of exploration vs. exploitation (Russell and Norvig, 2010, p. 839). The idea is that the agent should not always perform the best action at every time step (exploitation) and do sub-optimal actions to facilitate exploration of so-far unseen states. Given that the agent does not know ahead of time what good states are, it risks converging towards a sub-optimal policy if it has not seen enough different states. The agent needs to combine the exploitation of previous knowledge and explore new options to ensure that it finds a good policy. In other words, it is crucial that the model explores many different actions to be sure that it finds the best actions while at the same time exploiting the best actions enough times so that it finds strong sequences of actions and becomes confident in these actions.

A common way to achieve this is by deploying an ϵ-greedy strategy (Mnih et al., 2015). In this setup, the agent chooses a random action, over the best action with probability ϵ. ϵ will typically be a value that decreases over time, meaning that the agent deploys more aggressive exploration in the beginning, before steadily choosing the best actions more and more often. ϵ is decayed exponentially in this Thesis. Given a decay rate d, ϵ at time step t is calculated as follows:

$$\epsilon_t = \epsilon_{end} + \frac{\epsilon_{start} - \epsilon_{end}}{e^{t/d}}$$

2.3.14 Other Techniques

Some more machine learning techniques are mentioned in the context of the related work covered in Chapter 3. The main ones will now be *briefly* mentioned.

Naive Bayes

Naive Bayes (Russell and Norvig, 2010, p. 808) is a supervised learning algorithm based on Bayes Theorem and the so-called naive assumption that features are not correlated with each other (i.e., independent variables). It is a simple machine learning algorithm that has empirically shown to yield strong results, even though the naive assumption might not be strictly true for the given problem (Russell and Norvig, 2010, p. 499). It also has the advantage that it is fast and easily scaled (Alama et al., 2014).

k Nearest Neighbors

k Nearest Neighbors (k-NN) (Russell and Norvig, 2010, p. 738) is a method that, given a new example, computes the k most similar examples to the new example. This is based on some distance measure between points in the feature space. A way to make k-NN more sophisticated is by including a *weight* on the features. For example, one typically wants rare features to have a greater impact on the similarity measure, and common features to have less impact. A popular way to achieve this is by using so-called *term frequency–inverse document frequency* (TF-IDF). TF-IDF is a concept from the field of information retrieval that normalizes the weight of a feature based on how common the feature is across all examples.

TreeLSTM

TreeLSTM (Tai et al., 2015) is a method that can be used on tree structures. It is a generalization of the Long Short-Term Memory (LSTM) unit (Hochreiter and Schmidhuber, 1997), able to embed the topology of tree structures. This is done by having LSTM units (explained below) depend not only on the input vector and the hidden state at the previous time step but also on the hidden state of units belonging to children nodes in the tree. The idea is that this allows information to pass from children to the parent node, meaning that the embedding will capture the relationship

between these nodes. This is different from regular chained LSTM units that linearly pass information between tokens in a sequence.

LSTM units are a special case of recurrent neural networks (RNNs). In short, RNNs are neural networks containing nodes with edges that point back to the node (i.e., they are *recurrent*). That is, inputs to the recurrent node depend not only on the new input in the input layer but also on the previous inputs to the node. LSTMs are different from RNNs in that they also include three gates: input gate, output gate, and forget gate. By closing the input gate, new inputs will not override the LSTM unit information, making it capable of longer-term memory. The forget gate controls how long a sequence element should be part of the recurrent information in the unit, while the output gate controls the activation of the output from the unit.

Chapter 3

Related Work

The first section in this chapter, Section 3.1, will cover how the structured literature review for this Master's Thesis was performed. Section 3.2 addresses related work involving Auto-ITP. This section is considered the most relevant for the experiments in this Thesis. Then, Section 3.3 covers so-called Hammers, which is another popular way of automating ITP systems. This method is fundamentally different from Auto-ITP but still aims to achieve end-to-end automation of ITP systems. Hammers are also interesting because several machine learning techniques have been deployed in these systems to increase performance, and they have been used to supplement Auto-ITP models. Finally, Section 3.4 covers other related work involving machine learning applied to mathematics and formal reasoning. Although this literature addresses different problems than Auto-ITP, many subproblems overlap, and many techniques can be applied to Auto-ITP.

3.1 Literature Review

The initial interest for the topic in this Thesis came from an article written by a team at Google, where they introduced a new proof dataset (Kaliszyk et al., 2017). To explore the topic further, a top-down approach was used, where the related work section in Kaliszyk et al. (2017) served as a starting point. The literature review then quickly evolved into three distinct branches explored in parallel. The first was related work in the Auto-ITP space, the second was work on Hammers, and the third was a broader review of machine learning applied to formal reasoning. A review of ATP, ITP and machine learning was also conducted. This was performed on an ongoing basis, as important concepts were mentioned in literature. ATP, ITP and machine learning are considered background theory and therefore included in Chapter 2.

In order to review the current literature on Auto-ITP, the starting point was, as mentioned, the work of Kaliszyk et al. (2017). They pointed to other related work leveraging underlying ITP systems to train machine learning models, which in turn pointed to other related work within the Auto-ITP space. All Auto-ITP frameworks developed have been given a name, which was helpful because it enabled concrete searches on Google and Google Scholar for specific Auto-ITP frameworks. Such searches were very fruitful. For example, searching for "HOList" on Google revealed a website dedicated to this framework, where all Auto-ITP efforts in HOList

are summarized[1]. Moreover, searching for "CoqGym" on Google Scholar revealed work by several research groups using CoqGym as their benchmark (Sanchez-Stern et al., 2020; First et al., 2020). More general Google Scholar searches supplemented this. Here, keywords like "tactic prediction" and "automating proof assistants" were used, revealing relevant work outside of any defined Auto-ITP framework (Nawaz et al., 2020; Yang et al., 2016; Lee et al., 2020; Szegedy, 2020; Lample and Charton, 2020).

Hammers were researched similarly to Auto-ITP, with a top-down approach as the primary review method. This search started with HOL Light's Hammer: HOL(y)Hammer (Kaliszyk and Urban, 2015b). It became clear that there are many Hammer systems already developed. However, Hammers are mainly interesting as a comparison to Auto-ITP in this Thesis, and the review was therefore restricted to Hammer systems developed for ITP systems with an Auto-ITP counterpart. This led the review to focus on HOL(y)Hammer and its evolution (Kaliszyk and Urban, 2013, 2014, 2015b), and the Hammer system for Coq: CoqHammer (Czajka and Kaliszyk, 2018). Another essential resource on Hammers was the survey article Blanchette et al. (2016).

In order to fully understand Auto-ITP, it was necessary to have a basic grasp of the underlying ITP systems and the fundamentals of traditional ATP. Therefore a review of the ITP systems themselves was conducted. Two main approaches were taken. One was to read survey literature on ITP systems. These articles compared different modern ITP systems (Nawaz et al., 2019) and explained historical aspects of the systems (Harrison et al., 2014; Gordon, 2000). The second was to use the documentation provided by the ITP community[2][3][4]. ITP systems are relatively mature and have thorough documentation. As for ATP, the primary resource was first a survey lecture given by the ITP pioneer John Harrison[5], which pointed to relevant techniques and approaches in the field. The methods used by ATP systems are well studied and included in several books on logic and inference. The primary sources for this research were the widely adopted Russell and Norvig (2010) and Robinson and Voronkov (2001), as well as articles introducing the relevant techniques (Smullyan, 1968; Rusinowitch, 1991).

Machine learning literature was reviewed on an ongoing basis, as machine learning techniques came up in the Auto-ITP literature. Both introductory articles and survey articles were used. To filter out the most relevant articles, the supervisor for the Master's Thesis, Björn Gambäck, provided valuable insights and suggestions.

[1] https://sites.google.com/view/holist/home
[2] https://coq.inria.fr/distrib/current/refman/
[3] https://www.cl.cam.ac.uk/ jrh13/hol-light/reference.html
[4] https://hol-theorem-prover.org/#doc
[5] https://www.lektorium.tv/lecture/14805

3.2 Auto-ITP

In this Thesis, Auto-ITP refers to recent efforts by the machine learning community to build predictive models on top of existing ITP systems in order to predict tactic application (Bansal et al., 2019a; Yang and Deng, 2019; Gauthier et al., 2017). The whole system (ITP + predictive model) can automate the theorem proving task end-to-end, in a way where the machine learning model "acts as the human user". Figure 3.1 illustrates this setting. In order to be clear, the following definition is provided for Auto-ITP:

Auto-ITP *Any approach to automating an underlying ITP system where the proof search is driven forward by machine learning models that have learned to predict what tactics and tactic arguments to apply in a given proof state.*

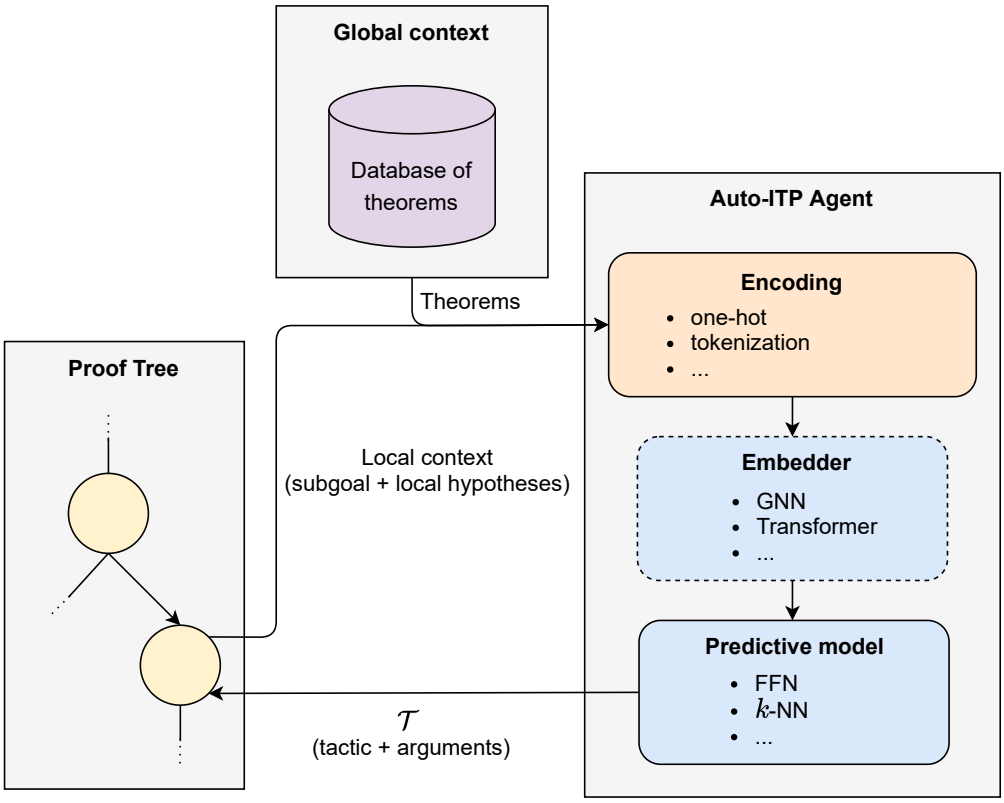


Figure 3.1: Overview of the Auto-ITP setting. Examples of possible encoding, embedding and predictive approaches are included for the Auto-ITP model.

There are multiple ways to train Auto-ITP models. In particular, one has to decide what kind of data the models are going to train on. In an *imitation* setting, the models are trained on human-written proofs (Bansal et al., 2019b). In this way, the model tries to imitate how humans prove theorems. In a *self-learning* setting models trains on its own proofs. These are typically generated during a reinforcement learning session (Bansal

et al., 2019b). It is possible to have a pure imitation setting, a combination of imitation and self-learning, and a pure self-learning setting.

The Auto-ITP process can be interleaved with Hammer calls. This means that the Hammer tries to prove the current subgoal first. If it fails, the Auto-ITP agent selects a new tactic, which leads to a new subgoal. The Hammer can then try to prove this subgoal, and so on. The same idea can be used with the ITP system's internal automatic engine (see Section 2.2.3).

Although formal reasoning is a cornerstone of symbolic-based artificial intelligence (Russell and Norvig, 2010), frameworks and benchmarks for combining machine learning and formal reasoning has been lacking. Auto-ITP research has been focused on providing full-fledged frameworks to address this. These frameworks generally choose an existing ITP system as the starting point and provide an Application Programming Interface (API) for engaging with the system programmatically. Machine learning researchers can then interact with the ITP systems in a black box fashion and avoid overheads associated with learning ITP specific domain knowledge. Four Auto-ITP frameworks have been developed so far. An overview is provided in Table 3.1. Auto-ITP models for each framework will now be covered. For the CoqGym framework, more details of the framework itself is provided.

Table 3.1: Overview of existing Auto-ITP frameworks. The data is gathered from several sources: Kaliszyk and Urban (2013); Gauthier and Kaliszyk (2015); Bansal et al. (2019a); Gauthier et al. (2017); Huang et al. (2019); Yang and Deng (2019). Values are rounded to the closest thousand.

Name	Underlying ITP	Human-written proofs
HOList	HOL Light	29k
TacticToe	HOL4	8k
GamePad	Coq	2k
CoqGym	Coq	71k

3.2.1 TacticToe

The first attempt at Auto-ITP was made by Gauthier et al. (2017), in the TacticToe environment. Gauthier et al. (2017) focus exclusively on k-NN as the core machine learning model. This is motivated by the success of this technique in the Hammer system HOL(y)Hammer[6] (explained later in Section 3.3.3). TacticToe extracts syntactic features of the current conjecture and scores its similarity to already proven conjectures using k-NN with an Inverse Document Frequency-based weighting scheme. Inverse Document Frequency essentially means that similarity is normalized based on how common terms are. The tactics used to prove the k most similar already-proven goals are applied to the

[6]The developers of TacticToe are also pioneers on Hammer research

current goal.

The first experiments with TacticToe did not treat argument prediction as a standalone problem. Instead, they simply predict tactics with already defined arguments (Gauthier et al., 2017). However, in later experiments, Gauthier et al. (2020) generalize tactics by removing their arguments. Arguments are then predicted by a separate k-NN model. Furthermore, they have one model predicting arguments from the global context and another model predicting arguments from the local context.

Gauthier et al. (2017) use a novel approach to proof search: a modified version of the best-first A* algorithm which pursues proof paths that are most likely to lead to a full proof. They do this by scoring each not-yet closed node in the search tree and choosing the node with the best score as the next node to expand. In order to score nodes, Gauthier et al. (2017) try a few different variations. The most successful is a simple summation of the depth of the node and the number of tactics previously applied on the goals in the node. Intuitively it makes sense that goals deeper in the proof tree are likely to be simpler than goals further up, and therefore easier to prove.

Gauthier et al. (2020) replace A* by a variant of the popular Monte Carlo algorithm (Raychaudhuri, 2008). In short, the next node to be evaluated is based on the number of times nodes along its proof path have been visited. An exploration term is also included, to allow nodes with not the highest score to be chosen from time to time.

Both the A* experiments (Gauthier et al., 2017) and the Monte Carlo experiments (Gauthier et al., 2020) complement the proof search by integrating a minimal version of a Hammer with TacticToe. This is done by invoking one of HOL4's (TacticToe's underlying ITP system) internal automatic engines (see Section 2.2.3) every time a selected goal is being evaluated. The hope is that the automatic engine will be able to prove the goal without further need for proof tree expansion. As noted by Gauthier et al. (2020), integration with Hammers are expensive and built-in an internal automatic engine are therefore chosen instead.

Results from Gauthier et al. (2017) and Gauthier et al. (2020) are summarized in Table 3.2. This constitutes state-of-the-art in TacticToe. Models are only trained in a supervised manner human-written proof (the Monte Carlo technique only updates the next-node heuristic in a reinforcement manner, not the machine learning model itself).

3.2.2 HOList

HOList is an Auto-ITP framework developed by Google (Bansal et al., 2019a). Initial results in this framework are provided by Bansal et al. (2019a). They have two networks responsible for embedding: one for the goal and one for arguments. They decide to drop the hypotheses from the local context, meaning that only arguments from the global context are considered. This comes at the cost that the model will fail on certain

Table 3.2: State-of-the-art and main results in TacticToe. Results are from experiments in Gauthier et al. (2017, 2020).

Setting	**Proof Search**	**Argument prediction**	**Hammer**	**Result**
Imitation	Monte Carlo	Yes	Automatic engine	**66.4%**
Imitation	A*	No	Automatic engine	39.43%
Imitation	A*	No	None	29.73%

goals while keeping the experiment reasonably simple. Bansal et al. (2019a) use a neural network called WaveNet van den Oord et al. (2016) to compute the embeddings. Details of WaveNet are omitted as it is not highly relevant for the rest of the Thesis.

The embedding for the goal is fed into a simple one-layer FFN. This network predicts what tactic to choose. For each global context theorem, its embedding is concatenated with the goal embedding and fed into another one-layer FFN. This network ranks the relevance of the theorem.

In Paliwal et al. (2020) the same high-level architecture is used. However, they choose to use Graph Neural Networks (GNNs) as embedding networks. In order to do this, they represent each term as an Abstract Syntax Tree (AST). A few AST modifications are tested. The most notable being:

- Standard AST.
- *Subexpression sharing.* Nodes that are syntactically equal are merged.
- *Top down.* Only keep edges from parent to children.
- *Bottom up.* Only keep edges from child to parent.

Paliwal et al. (2020) choose GNN message passing over the similar TreeLSTM architecture, because TreeLSTM fail to consider the full *context* of sub-expressions. TreeLSTM will always compute the same embedding for a sub-expressions, regardless of its context in the whole formula, as they only consider information flow from children nodes to parent nodes, not in both directions. Part of the reason why Paliwal et al. (2020) experiment with top-down and bottom-up variations of the ASTs is to better understand what constitutes the most important context for sub-expressions (e.g., does the parent node contain more important semantic meaning for child nodes than the other way around?).

Bansal et al. (2019a) and Paliwal et al. (2020) train in both a strict imitation manner, in an imitation+self-learning manner, and in a strict self-learning manner. Self-learning is achieved by appending newly generated theorems to the training set. A hybrid imitation and self-learning setting is done by seeding the models with human-written proofs first, allowing it to train on that data in addition to

machine-generated proofs. In the strict self-learning setting, the dataset is initially empty.

Results for the self-learning setting were improved in Bansal et al. (2019b), by including a mechanism for doing exploration of possible tactic arguments, instead of only exploiting the top-ranked premises. In short, "explorative" premises are selected based on an Inverse Document Frequency-style similarity measure, similar to the k-NN weighting scheme used by Gauthier et al. (2017). Top-ranked premises and exploration premises are interleaved into one list and sent as arguments with the tactic. If the model produces a successful proof, the self-learning loop ensures that this proof is used for further training, meaning that the model can explore different (and maybe unconventional) tactic arguments.

Main results from experiments in Bansal et al. (2019a,b); Paliwal et al. (2020) are summarized in Table 3.3, which constitutes state-of-the-art in the HOList framework. All experiments use Breath-First Search to traverse the proof tree (i.e., decide the next node to evaluate), and no Hammer or automatic engine calls are interleaved with the proof procedure. Bottom up ASTs perform significantly worse than other AST variations. This indicates that information flow from parent to child node is important to capture in AST embeddings. Furthermore, imitation+self-learning models preform better than both strict imitation and self-learning models.

Table 3.3: State-of-the-art and main results in HOList. Results are from experiments in Bansal et al. (2019a,b); Paliwal et al. (2020).

Setting	Embedding	AST variant	Exploration	Result
Imitation	WaveNet	-	No	32.65%
	GNN	Standard	No	46.66%
	GNN	Bottom up	No	41.86%
	GNN	Top down	No	48.40%
	GNN	Sub. share	No	**49.95%**
Imitation + Self-learning	WaveNet	-	No	38.9%
	GNN	Sub. share	Yes	**59.9%**
Self-learning	GNN	Sub. share	Yes	**56.3%**

3.2.3 GamePad

Researchers at OpenAI and Berkeley introduced the GamePad framework in 2019 (Huang et al., 2019). However, there have yet to be any models performing end-to-end theorem proving in this framework. Part of the reason for this is likely that only 1,602 theorems are available in this framework.

Unlike in other Auto-ITP experiments, Huang et al. (2019) do not attempt to

prove theorems end-to-end. Instead, they introduce three proxy metrics: *position*, *tactic* and *argument* prediction. Position prediction is the task of predicting how many tactic applications are left before the current goal/subgoal is proved. Tactic prediction is the task of predicting the tactic used in the human-written proof. Argument prediction is the task of predicting what arguments were used with a given tactic, in the human-written proof.

Like with experiments in HOList, Huang et al. (2019) simplify the argument problem by only focusing on the global context. In addition, they use a preprocessing step that generalizes tactics. This results in only 23 tactics to predict. Furthermore, position prediction is simplified to predicting one of three classes: (1) close (< 5 steps), (2) medium (between 6 - 19 steps), and (3) far (> 20 steps).

Notable methods used by Huang et al. (2019) include: (1) always guess the most common category, (2) predict using a straight-forward Support Vector Machine (SVM) method, and (3) using TreeLSTM and FFN. For (2), formulas are not embedded in any feature space. Instead, metrics like goal size and the number of local assumptions are used as features. For (3), formulas are embedded using TreeLSTM on the Coq AST representations before a FNN makes the final prediction. Results from experiments in (Huang et al., 2019) are summarized in Table 3.4. FFN+TreeLSTM outperforms the other models on all tasks. Results indicate that argument prediction is a difficult task, with an accuracy of 23.91% being the best result from Huang et al. (2019).

Table 3.4: State-of-the-art and main results in GamePad. Results are from experiments in Huang et al. (2019)

	Position	**Tactic**	**Argument**
Most common category	53.66%	44.75%	$< 10\%$
SVM	57.52%	49.45%	$< 10\%$
TreeLSTM+FFN	**66.30%**	**60.55%**	**23.91%**

3.2.4 CoqGym

Yang and Deng (2019) developed the Auto-ITP framework CoqGym in 2019. The framework implements a Python API for interacting with Coq, and provides a large set of proof data. CoqGym's proof data comprises 70,856 human-written theorems from 123 different formalization projects in Coq. CoqGym will be explained in more detail, as it is the primary framework concerned in this Master's Thesis.

Dataset

The dataset is split between a train, validation, and test set. This includes proof data belonging to both pure mathematical domains and software and hardware verification.

CoqGym also ships with *synthetic* proofs, generated from human-written theorems. These proofs were generated from intermediate subgoals found in the human-written proofs. Yang and Deng (2019) extract synthetic proofs of length (i.e., the number of tactics used in the proof) 1, 2, 3, and 4. The process of generating synthetic theorems convert terms in the subgoals into hypotheses in the local context, before human-written sequences of tactics are applied followed by the `auto` tactic. The sequence of human-written tactics can be of desired length to generated fixed-size synthetic proofs. Crucially, this process makes synthetic proofs similar to human-written proofs meaning they are not a replacement for reinforcement learning but rather an enhancement to human-based imitation training. Table 3.5 summarizes the proof data in CoqGym for the train, validation, and test split.

Table 3.5: The CoqGym dataset. `h` and `s` refer to human-written and synthetic proofs, respectively. Information on the number of synthetic proofs for the different splits is not provided by Yang and Deng (2019).

	`h`	`s`, length 1	`s`, length 2	`s`, length 3	`s`, length 4
Train	43,844	-	-	-	-
Validation	13,875	-	-	-	-
Test	13,137	-	-	-	-
Total	**70,856**	**159,761**	**109,602**	**79,967**	**61,126**

SerAPI

In order to communicate with Coq, CoqGym leverages an API called *SerAPI* (Gallego Arias, 2016). SerAPI responds with s-expressions (i.e., a nested list of symbols with an obvious tree representation), representing the Coq response for the given input. However, CoqGym wraps the SerAPI calls in a Python class, meaning that one does not have to directly deal with SerAPI when developing Auto-ITP models in CoqGym.

SerAPI calls are time-consuming and the main bottleneck when a model proves theorems in an interactive mode (Yang and Deng, 2019). CoqGym sets a default timeout for each SerAPI call to 12 minutes. If one needs a model to cover more proofs in a shorter amount of time (e.g., when training a reinforcement learning agent), this timeout parameter can be modified.

Abstract Syntax Trees

All Coq expressions in CoqGym have an associated AST representation. These representations are built using a fixed vocabulary of *nonterminals*. Simply put, the nonterminals define the values in which a node in the AST can take. There are 55 nonterminals in CoqGym. They allow an unambiguous and general way to build ASTs from Coq

expressions. To build ASTs, CoqGym uses a Python library called Lark[7], which parses s-expressions based on a provided well-defined grammar.

Results in CoqGym

For initial testing in CoqGym, Yang and Deng (2019) develop a deep learning model capable of generating tactics in a non-trivial way. They call their model *ASTactic*. The main idea is to generate tactics as ASTs, not predict tactics from a pre-defined set. To do this, ASTactic leverages the tactic space defined by Coq's context-free tactic grammar. This grammar is briefly explained in Section 2.2.5. More details can be found in (Barras et al., 1997), for interested readers.

ASTactic embeds ASTs using TreeLSTM. The embeddings and features from the tactic grammar are inputted to a Gated Recurrent Unit (GRU)[8]. The GRU is responsible for building a new AST, which represents the next tactic to apply. The hidden state of the GRU $\boldsymbol{s_t}$ is the central component in expanding the tactic AST. $\boldsymbol{s_t}$ is updated based on $\boldsymbol{s_{t-1}}$, and a concatenation of the current node's symbol, the parent node's symbol, the production rules from the context-free tactic grammar, the goal embedding and the weighted sum of possible tactic arguments. Arguments are weighted using an attention mechanism, which depends on the argument and $\boldsymbol{s_{t-1}}$. Yang and Deng (2019) also test Coq's Hammer CoqHammer on CoqGym's test set, and experiment with interleaving calls to CoqHammer with ASTactic's proof procedure.

Another Auto-ITP model in CoqGym is TacTok (First et al., 2020). TacTok was motivated by the fact that the not-yet-finished proof contains semantic information in the proof procedure. This makes it useful in predicting the next tactic to apply to subgoals in the node. First et al. (2020) follow the same architecture as ASTactic; they generate tactics as ASTs, based on embeddings of context and goals, in addition to the tactic space. However, in TacTok, the current path of tactics in the proof tree (i.e., the unfinished proof) to the current node is also embedded and part of the GRU input.

Proverbot9001 (Sanchez-Stern et al., 2020) is another model built to do Coq Auto-ATP. The model works by dealing with tactic selection and premise selection separately. Formulas are embedded using Recurrent Neural Networks (RNNs). Tactics are predicted by inputting the embedding of the local context to an FFN, resulting in a ranking of a pre-defined set of tactics. For each possible tactic argument, a score is computed by another FNN. This setup is similar to models in HOList (Bansal et al., 2019a). Tactics and arguments are then given a common score by multiplying their scores. The unique thing about this is that arguments themselves influence if a core tactic should be chosen or not instead of first finding the best tactic and then computing the most relevant arguments for it.

[7] https://lark-parser.readthedocs.io/en/latest/
[8] GRUs are almost like LSTMs. The main difference is that GRUs do not have output gates.

The main results from the experiments in Yang and Deng (2019); First et al. (2020); Sanchez-Stern et al. (2020) are shown in Table 3.6. Yang and Deng (2019) and (First et al., 2020) run their experiments on the whole dataset in CoqGym, while Sanchez-Stern et al. (2020) only run them on a one specific Coq library: the `CompCert` project. All experiments used Depth-First Search to traverse the proof tree.

Table 3.6: State-of-the-art and main results in CoqGym. Results are from experiments in Yang and Deng (2019); First et al. (2020); Sanchez-Stern et al. (2020)

	Dataset	**Hammer**	**Result**
ASTactic	Full	None	12.2%
TacTok	Full	None	**12.9%**
CoqHammer	Full	-	24.8%
ASTactic	Full	CoqHammer	**30.0%**
CoqHammer	`CompCert`	-	7.39%
ASTactic	`CompCert`	None	4.59%
Proverbot9001	`CompCert`	None	**19.36%**

3.3 Hammers

Hammers (Blanchette et al., 2016) are a way of achieving automation of ITP systems. This means they can serve as a comparison to Auto-ITP models. Furthermore, they can enhance the ability of Auto-ITP models by interleaving calls to the Hammer with Auto-ITP's automated tactic application. However, how Hammers achieve automation is different from Auto-ITP. Therefore, they are considered a distinct topic in this Thesis. This section will describe Hammers and provide an overview of results from two concrete Hammers: HOL(y)Hammer (Kaliszyk and Urban, 2013) and CoqHammer (Czajka and Kaliszyk, 2018). These are the most relevant for this Thesis because they are the Hammers developed for HOL Light, HOL4, and Coq, and therefore provide automation of the same ITP systems as current Auto-ITP models do.

Hammers are tools built on top of existing ITP systems, allowing the system to prove theorems automatically. They do this by outsourcing the theorem proving job to third-party ATP systems. Today, most ITP systems have a Hammer extension. After the first Hammer appeared, in the form of *Sledgehammer* for the ITP system Isabelle (Böhme and Nipkow, 2010), the field experienced traction, and other ITP system designers enabled their own system with a corresponding tool. Hammers are particularly popular for users who are formalizing large proofs, as they allow much of the task to be automated. For example, Kaliszyk and Urban (2015c) were able to generate proofs for 40% of the theorems in the famous Mizar Mathematical Library (see Grabowski et al. (2010) for details on the Mizar Mathematical Library,

and Bancerek et al. (2018) for its role in ITP research) fully automatically using Hammers.

An important detail to note is that Hammers do not have to be used in a way that achieves full automation. One can, for example, invoke the Hammer to solve particular subgoals in the proof tree, while a human is still responsible for guiding the system to those subgoals. Full automation is only achieved when the Hammer is already invoked on the top-level goal. This means that the proof procedure carried out by a Hammer does not generate a proof tree, in the way shown in Figure 2.4.

3.3.1 The 3-step Process

Blanchette et al. (2016) explain the ideas that allow Hammers to work. Hammers are made possible by a 3-step process:

1. *Premise selection.* Select a subset of available theorems to pass along with the goal conjecture to the ATP systems. This is equivalent to deciding the Knowledge Base for the ATP systems to use (see Section 2.1).

2. *Logic translation.* For the ATP systems to function, they need to have both the goal conjecture and Knowledge Base in a logic they can understand. Therefore a process of translating from the ITP system's logic (usually a variation of Higher-Order Logic) to the ATP system's logic (usually a variation of First-Order Logic) is needed.

3. *Proof reconstruction.* For a proof found by an ATP system to be accepted by the ITP system, it is necessary to reconstruct the proof so that the ITP system can check it. This is because the translation step is usually not a fully sound process. The reconstructed proof is checked by running it through the ITP system's kernel (see Section 2.2.4).

These problems have to be solved sequentially. The high-level architecture of a Hammer system is depicted in Figure 3.2.

When evaluating Hammers a *human-chronological* approach is generally used (Blanchette et al., 2016). This refers to a setting where the Hammer tries to build the corpus of proof data in the same order humans built it. In other words, at any given time, when the Hammer is proving a conjecture c, the only theorems available to the Hammer are the ones that were also available to humans, when they proved conjecture c. In this way, the Hammer emulates the actual corpus construction. This is a way to assure that the correctness of the Hammer evaluation is guaranteed by construction. It also allows the Hammer to prove a corpus in a "push-button" mode. This is different from a typical machine learning setting, where the model is usually indifferent to the order in which it learns from examples. This means that results from a Hammer cannot be directly compared to the results of an Auto-ITP system.

In general, it is the premise selection step, together with the number of theorem provers and the resources provided (e.g. time constraints, CPU power, etc.), that determine the success of the Hammer, not the translation or the reconstruction steps. Translation and reconstruction are necessary steps with minimal ability to be optimized. Most Hammer research has therefore focused on premise selection. Premise selection is also highly agnostic to the underlying systems and can therefore be researched independently of a specific Hammer system.

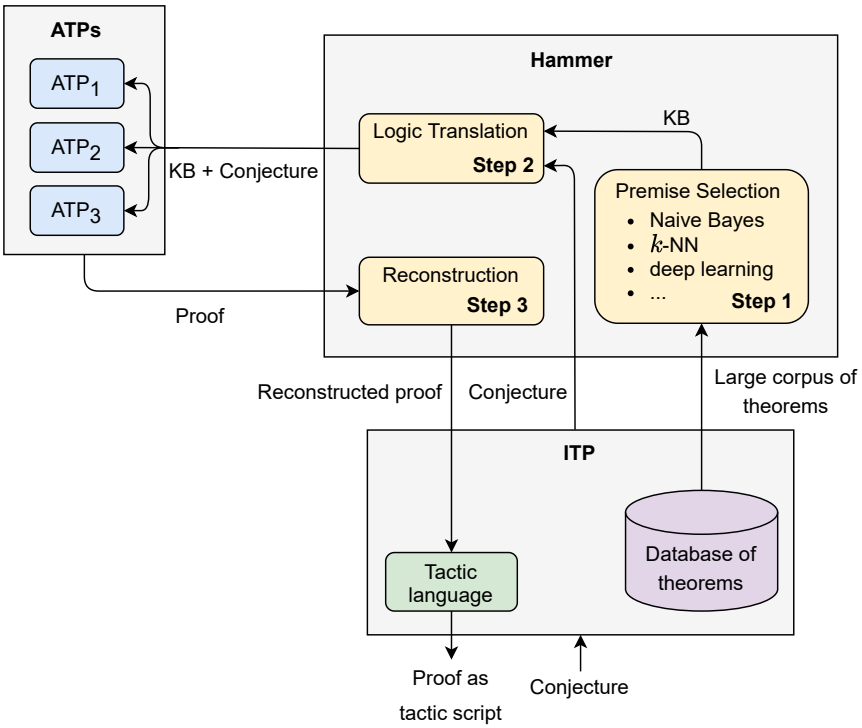


Figure 3.2: The high-level architecture of a Hammer system. KB denotes Knowledge Base.

3.3.2 Premise Selection

Premise selection is the process of choosing a subset of available theorems to include as background theory in a proof procedure. Alama et al. (2014) define it in the following way:

Premise selection *Given a large number of premises P and a new conjecture c, predict those premises from P that are likely to be useful for automatically constructing a proof of c* (Alama et al., 2014).

Hammers do this step outside of the ATP systems. However, it is common for ATP systems to have their own internal premise selector, meaning that premise selection is

performed twice in a Hammer: external of the ATP systems and internally within the ATP systems.

Several techniques have been applied to the problem, including both non-learning and learning-based methods. In particular, k-NN, Naive Bayes, Kernel-based methods, and deep learning methods have been applied to premise selection[9] (Alama et al., 2014; Kaliszyk et al., 2017; Wang et al., 2017). So far, only Naive Bayes and k-NN have been implemented as part of a full-fledged Hammer. Naive Bayes and k-NN models are easy to implement and, crucially, scale well with a full-fledged Hammer (Kaliszyk and Urban, 2014). A popular non-learning method is the *SInE* method (Hoder and Voronkov, 2011). This is often used internally in ATP systems. Kühlwein et al. (2012) show in a comparison of different premise selection methods that non-learning methods perform significantly worse than learning-based methods when tested outside of ATP systems.

3.3.3 HOL(y)Hammer and CoqHammer

Kaliszyk and Urban (2014) were the first to experiment with a Hammer for HOL Light. This resulted in the Hammer system HOL(y)Hammer[10]. Gauthier and Kaliszyk (2015) ported the same Hammer to make it work for HOL4. In other words, HOL(y)Hammer is the Hammer used by both HOL Light and HOL4.

For HOL(y)Hammer's premise selection method to work, features from the theorems in the proof need to be extracted. So far in Hammer research, only syntactic features have been used.

Kaliszyk and Urban (2013) test HOL(y)Hammer on one of HOL Light's proof libraries called `Flyspeck`, and Gauthier and Kaliszyk (2015) test it on HOL4's Standard Library (SL). The Hammer is always tested in a human-chronological way, and the premise selector is only trained on top-level human-written proofs. Kaliszyk and Urban (2013) test HOL(y)Hammer using Naive Bayes and k-NN as the premise selector. Gauthier and Kaliszyk (2015) only test it using k-NN.

Czajka and Kaliszyk (2018) develop the Hammer system CoqHammer for Coq. Most parts of CoqHammer function in the same way as HOL(y)Hammer. Of course, because of different logical foundations, the details of the translation and reconstruction steps must be adopted to account for this. Premise selection in CoqHammer has also only been implemented using k-NN and Naive Bayes.

State-of-the-art for both HOL(y)Hammer and CoqHammer are shown in Table

[9] Not all of these methods are explained in the background theory, as not all are highly relevant for the Thesis.

[10] https://www.thibaultgauthier.fr/holyhammer.html

3.7. E[11] and Vampire[12] are ATP systems. Note that *E-BliStr* denotes E run with a mechanism that allows it to predict what inference strategy (e.g., resolution, tableaux, superposition calculus, etc.) best suited for solving the given conjecture.

Table 3.7: State-of-the-art and main results for HOL(y)Hammer and CoqHammer. Results are from experiments in Kaliszyk and Urban (2013); Gauthier and Kaliszyk (2015); Czajka and Kaliszyk (2018). SL denotes the ITP system's Standard Library.

Hammer	Premise Selection	#Premises	Dataset	ATP	Result
HOL(y)Hammer	k-NN	128	`Flyspeck`	E	31.43%
HOL(y)Hammer	k-NN	128	`Flyspeck`	E-BliStr	**34.88%**
HOL(y)Hammer	Naive Bayes	164	`Flyspeck`	E	24.17%
HOL(y)Hammer	k-NN	128	HOL4's SL	E-BliStr	**44.45%**
CoqHammer	k-NN	1,024	Coq's SL	Vampire	**28.82%**
CoqHammer	k-NN	1,024	Coq's SL	E-BliStr	25.59%
CoqHammer	Naive Bayes	256	Coq's SL	E-BliStr	17.50%

3.4 Other Applications of Machine Learning in Formal Reasoning and Mathematics

Although this Thesis is mainly concerned with Auto-ITP, numerous works address different uses of machine learning applied to formal reasoning and mathematics. This section will provide an overview of selected literature review findings that do not fit neatly into Auto-ITP or Hammers, but are still interesting as they can inspire new approaches in Auto-ITP research. Indeed, as will become apparent in Section 4.1, some of the ideas in the following subsections provide natural starting points for this Thesis' research in CoqGym.

3.4.1 Transformer Models Applied to Mathematics

A handful of recent papers address the idea of applying Natural Language Processing (NLP) methods in both formal and informal mathematics and can inspire potential Auto-ITP models. Three relevant papers are discussed here, all revolving around the Transformer model (Vaswani et al., 2017).

Rabe et al. (2020) introduce a so-called *skip-tree* task. The idea is to pre-train a Transformer model on a task similar to language models such as BERT (see Section 2.3.12). This is achieved by having the model predict *masked* sub-expressions in the AST representation of logical expressions based on the unmasked part of the sub-expression.

[11] https://wwwlehre.dhbw-stuttgart.de/ sschulz/E/E.html
[12] http://www.vprover.org/

Rabe et al. (2020) use expressions from the HOList dataset for training, meaning that the expressions take the form of Higher-Order Logic. The pre-trained model is then tested on a series of mathematical tasks, with no fine-tuning to specialize it on each particular task. Although it is hard to argue how strong the results are – there are no benchmarks to compare them to – they seem to indicate solid formal reasoning capabilities. For instance, the model achieves an accuracy of 40.86% when tasked with predicting missing local context hypotheses, and as high as 96.23% when the model predicts masked term types (i.e., what Higher-Order type a given term in the expression is).

Lample and Charton (2020) train a Transformer network to solve integral problems and ordinary differential equations (ODEs). However, unlike all related work discussed so far, Lample and Charton (2020) train on mathematics outside of any formal framework. Instead, they focus on much more familiar textbook-style syntactical expressions. The model does not train on the skip-tree task presented by Rabe et al. (2020), but rather on mathematical expressions expressed using infix notation. The Transformer decoder outputs the integral of the expression or solution to the ODE. The model is not pre-trained but still able to show strong results. For instance, the model is capable of solving of 81.2% of order two ODEs from the test set correctly.

OpenAI develops the theorem proving model GPT-f for the ITP system MetaMath (Polu and Sutskever, 2020). MetaMath (Megill and Wheeler, 2019) is an ITP system with a unique style of interaction. Instead of relying on high-level tactics, MetaMath uses a substitution-based proof procedure. In short, proofs search is only driven forward by substituting terms in the current goal using previously proved theorems[13]. GPT-f is a Transformer model, training in both a pre-training session and fine-tuned for MetaMath theorem proving. Pre-training is done on data collected from Github[14], Math StackExchange[15] and arXiv Math[16].

3.4.2 Synthesizing Theorems

Synthesizing theorems is the task of teaching machines to generate synthetic theorems automatically Wang and Deng (2020). This topic tackles data scarcity in the proof domain. It is related to self-learning Auto-ITP models; machine-generated theorems can be used in a feedback loop for such models. This is similar to how machine-generated proofs are used in HOList's self-learning models (Bansal et al., 2019b).

Generating synthetic theorems is an extensive topic in its own right. Only the example of Wang and Deng (2020) is mentioned here. They propose a setup where the goal is to generated human-like theorems in the MetaMath system (mentioned above).

[13] Note that, because MetaMath operates with such a distinct theorem proving setup, it is considered different from Auto-ITP in this Thesis.
[14] github.com
[15] https://math.stackexchange.com/
[16] https://arxiv.org/archive/math

The setup is based on loss functions used to teach models to generate new theorems. The loss function computes how "different" a machine-generated theorem is, compared to human-written theorems. An adversarial-based loss generated by a network trained to distinguish human-written and randomly generated theorems is proposed, in addition to a cross-entropy loss outputed by a language model trained on human-written proofs. In this way, models can be trained to gradually generate more and more human-like theorems.

3.4.3 Tactic Application in Latent Space

Another interesting, but still very infant, line of research is to predict the outcome of tactic applications using machine learning. Lee et al. (2020) provide some initial results. A model is trained to predict the resulting goal after the series of `rewrite` tactics are applied to the original goal. Although results are very much early stage, it is still a fascinating approach to tactic application, where the underlying ITP system is taken completely out of the loop.

3.4.4 Evolutionary Algorithms

A completely different approach to Auto-ITP is to use evolutionary algorithms for tactic prediction. Some initial results in this line of research are developed by Nawaz et al. (2020) and Yang et al. (2016). So far, no experiments have been conducted where end-to-end theorem proving is done on large corpora of proof data. This Thesis chooses not to focus on evolutionary algorithms simply because of time constraints, but it is an interesting avenue worth mentioning.

3.4.5 Internal Guidance

Internal guidance is an approach to ATP, where machine learning models guide the internal inference process. It has mainly been studied for analytic tableaux (see Section 2.1.2) style theorem provers (Urban et al., 2011; Kaliszyk and Urban, 2015a). The idea is to use machine learning models to predict which branch of the tableau to expand next and select a relevant subset of the Knowledge Base (i.e., premise selection). This speeds up the inference process by only considering the most promising branches and relevant theorems (Urban et al., 2011). (Loos et al., 2017) deploy such models in the ATP system E. Using deep learning internal guidance, they prove 7.36% new theorems in the famous Mizar Mathematical Library (see Grabowski et al. (2010) for details on the Mizar Mathematical Library).

This Thesis does not focus on internal guidance. However, it is an interesting topic to compare to Auto-ITP against and therefore mentioned. While both internal guidance and Auto-ITP revolve around machine learning applied to formal reasoning, they are, in some sense, on opposite sides of the spectrum. Internal guidance models prove theorems by guiding low-level inference processes far from how humans reasons

while Auto-ITP, on the other hand, operates on high-level tactics much closer to how humans reasons about mathematics (Yang and Deng, 2019).

3.4.6 Autoformalization

Szegedy (2020) outlines a possible path forward for developing better machine learning models in the context of formal reasoning and, more broadly, artificial general intelligence. He emphasizes the task of *autoformalization* as a critical ingredient for reaching artificial intelligence models capable of more generalized reasoning. The main idea is that the agent needs to take informal data as input, formalize this data in a way that is consistent with some logical framework, and then reason based on the formal data. Auto-ITP is a line of research targeting the latter – reasoning over formal data.

Szegedy (2020) argues that in order to perform autoformalization effectively, the model needs strong NLP and computer vision capabilities. While it is not the goal for this Thesis to do autoformalization, the vision outlined by Szegedy (2020) puts the relative niche task of Auto-ITP into an interesting context of generalized reasoning.

Chapter 4

Motivation, Agent Design and Architectures

With both background theory and relevant work having been presented, the reminder of this Master's Thesis is focused on new experiments in CoqGym. This chapter begins with a section about the motivation for various decisions made when scoping experiments and choosing deep learning techniques (Section 4.1). Then, a section is dedicated to the tactic group proxy metric experiment designed for CoqGym (Section 4.2). This is followed by a section about the new Interactive Theorem Proving (ITP) agent developed in this Thesis (Section 4.3). Finally, the last section describes the overall deep learning model architectures (Section 4.4). Some statistics from the CoqGym dataset are included in various parts of this chapter to understand the design decisions better.

4.1 Motivation

4.1.1 Choosing an Auto-ITP Framework

CoqGym has a larger dataset of human-written proofs than other frameworks, with ~42k more theorems than the following largest dataset (an overview of dataset sizes is shown in Table 3.1). Having lots of training data is hugely important when training deep learning models, and was therefore emphasized when choosing an Auto-ITP framework. CoqGym also provides a large number of synthetic proofs (explained in Section 3.2.4). No other framework does this.

Another consideration was the diversity of the research groups using each Auto-ITP framework. CoqGym is the only framework used as a benchmark by researchers outside the group that introduced the framework. In the case of CoqGym, both TacTok (First et al., 2020) and Proverbot9001 (Sanchez-Stern et al., 2020) compare themselves to ASTactic (Yang and Deng, 2019). This seems attractive, as it indicates other research groups finding CoqGym to be a good benchmark.

The adoption of the underlying ITP system was also considered. As inexperience with ITP systems was a concern in the early phase of the Thesis work, having a mature community of developers and users supporting the underlying ITP system

was important. Coq is one of the most widely adopted ITP systems. Because HOL Light, HOL4, and Coq are all open source projects hosted on GitHub, one can look at repository statistics to indicate the adoption of each system. Table 4.1 summarizes some main statistics. This is, of course, only an heuristic and not in any way indicating that one system is superior to others.

Table 4.1: GitHub repository statistics for HOL Light, HOL4, and Coq. Extracted April 30, 2021.

System	Contributors	Stars	Forks
HOL Light	8	276	54
HOL4	56	422	80
Coq	194	3,287	498

4.1.2 Usefulness of Proxy Metrics

It became clear early on when experimenting in CoqGym that the overhead of training a model to do end-to-end theorem proving is significant. This means that prototyping a model becomes cumbersome if the goal is end-to-end proving right away. A possible way to overcome this is by first using proxy metric experiments before moving on to end-to-end theorem proving.

Huang et al. (2019) already focus on this task in the GamePad framework. The experiments in Huang et al. (2019) are more straightforward than end-to-end theorem proving, while, as argued by Huang et al. (2019), they still have a clear similarity to the task of predicting tactic application. This makes them helpful in understanding a model's ability to do formal reasoning. In other words, it is likely hard for a model to perform poorly on the proxy metric tasks, but well on end-to-end theorem proving, and vice versa. This idea is followed here. A proxy metric experiment, based on predicting *tactic groups*, is introduced, with the goal that this will allow easier and faster prototyping of Auto-ITP models.

4.1.3 Machine Learning Interpretation of ITP Systems

The Auto-ITP models described in Section 3.2 interpret the ITP theorem proving task in different ways. TacticToe and HOList models turns the task into classification problems (Bansal et al., 2019a; Gauthier et al., 2020), where one model focuses on core tactics and another focuses on arguments. This is similar to ProverBot9001 (Sanchez-Stern et al., 2020). Together the models form a theorem proving agent. However, while core tactic prediction is a multi-class classification problem, argument prediction is viewed as a series of binary classification tasks. Moreover, the local context is typically discarded. Yang and Deng (2019) propose a different setup, involving building tactics from Coq's context-free tactic grammar. Core tactic prediction and argument prediction is intertwined in this

setup. While it is a flexible setup, allowing the models to express the full Coq tactic space, it is also more difficult to interpret as a traditional machine learning problem. Having an easy machine learning interpretation of the ITP theorem proving process is desirable, as it allows less ITP specific knowledge needed when researching Auto-ITP. This is the motivation for designing a new theorem proving agent in this Thesis.

4.1.4 Choosing Machine Learning Techniques

As shown by Paliwal et al. (2020), using Graph Neural Networks (GNNs) for embedding HOL Light expressions increases performance in HOList. Paliwal et al. (2020) argue that this is because such embedding techniques capture more of the semantic information contained in the expressions. This is also partly the reason why TreeLSTM is chosen for ASTactic (Yang and Deng, 2019). The success of GNNs in HOList serves as a significant motivation for trying the same approach in CoqGym.

With an easy-to-use proxy metric at hand, several GNN variations can be tested relatively quickly. Three message passing implementations will be used: (1) Multilayer Perceptron (MLP)[1], (2) Graph Convolutional Networks (GCN) (Kipf and Welling, 2017), and (3) Simple Graph Convolutions (SGC) (Wu et al., 2019). See Section 2.3.11 for an introduction to these GNN techniques.

(1) is directly inspired by the implementation in Paliwal et al. (2020), where MLPs are used for message passing. It also serves as comparison with the more sophisticated convolution-based techniques.

(2) is motivated by GCNs success on several graph tasks (Kipf and Welling, 2017).

(3) is motivated by the fast training allowed by SGC (Wu et al., 2019) and its strong performance on several graph tasks.

For end-to-end theorem proving, efforts are mainly focused on one architecture to make experiments more manageable. The DGCNN architecture (Zhang et al., 2018) (explained in Section 2.3.11) is a universal end-to-end graph classification architecture showing strong performance on several graph tasks and therefore serves as the basis for the implementation.

In a similar vein, the use of Transformers has shown promising results on several mathematical tasks (Rabe et al., 2020; Lample and Charton, 2020; Polu and Sutskever, 2020) (explained in Section 3.4.1). However, they have yet to be tested on the Auto-ITP task. Because of time constraints, developing a tailored Transformer architecture and performing pre-training, like the skip-tree task (Rabe et al., 2020) or the GPT-f pre-training (Polu and Sutskever, 2020) is left for future work. Instead, the powerful BERT (Devlin et al., 2018) architecture will be used off-the-shelf. BERT has shown state-of-the-art results on several NLP tasks (Devlin et al., 2018), and it is

[1] An MLP is essentially the same as an FFN. MLP is used here to avoid confusion with other models.

interesting to see how this model performs when natural language is substituted for formal expressions. To also see if there is any transferability between NLP pre-training and the Auto-ITP task, BERT will be tested both with and without pre-trained weights. These weights are provided by Devlin et al. (2018) after BERT has been trained on the NLP-specific tasks explained in Section 2.3.12.

Bansal et al. (2019b) hypothesize that it could potentially be easier for a machine learning model to teach itself to prove theorems rather than rely on imitation training. It might be the case that there is so much variation in the way humans prove theorems in Coq that it is difficult for the model to pick up on clear correlations. Also, Wang and Deng (2020) point out that the scarcity of labeled examples is a bottleneck in the formal reasoning domain. This motivates CoqGym experiments involving self-learning. A reinforcement learning agent is built with the idea that it can find its own "unique" style of proving theorems based on trial-and-error. The agent will also have a chance to learn from both successful and failed proof attempts. In other words, CoqGym's training data is used more efficiently, combating data scarcity.

In addition, Huang et al. (2019) have argued that theorem proving through an ITP system is similar to a game, in which the agent has a space of actions it can perform in any given state, where a subset of actions might lead to a better state. Reinforcement learning agents have shown strong performance on classical games, such as chess (Silver et al., 2018), making it a natural approach to also test in the context of Auto-ITP.

It turns out that running even a single theorem proving "game" in CoqGym is relatively expensive. Each action has to communicate with Coq via SerAPI (see Section 3.2.4), which is a time-consuming process. As explained by Yang and Deng (2019); ASTactic is tested on a CPU, rather than a GPU, because the neural network component of the agent is not the bottleneck but rather the SerAPI calls. This means that developing a reinforcement learning agent dependent on running lots and lots of theorem proving simulation is not ideal. Therefore, the famous Monte Carlo algorithm (Raychaudhuri, 2008) will not be used, but rather Q-learning (Russell and Norvig, 2010). Moreover, because both the proof space and tactic space is large, a *deep* Q-learning (Mnih et al., 2015) approach will be used, allowing the model to approximate the Q-function rather than encoding it explicitly (which would likely be infeasible).

4.2 Proxy Metric: Tactic Groups

The main motivation for having a proxy metric experiment is to allow faster and easier prototyping of models in CoqGym, with the idea being that promising models can eventually be used as full-fledged Auto-ITP models. A proxy metric would therefore ideally possess the following two characteristics:

1. The proxy metric should be easy and fast to use.

2. The proxy metric should be indicative of end-to-end theorem proving performance.

The proxy metric proposed here is based on individual proof steps extracted from the train and validation set in CoqGym. Yang and Deng (2019) provide a pre-implemented method that extracts such proof steps from human-written proofs. Extending this to also include synthetic proofs results in the proof step dataset shown in Table 4.2.

Table 4.2: Proof steps in CoqGym for both human-written and synthetic proofs.

	Human	**Synthetic**	**Total**
Train	121,644	174,076	295,720
Validation	68,180	113,048	181,228

The first simplification made for the proxy metric is to only consider the core tactics. Instead of predicting full tactic applications (i.e., tactic + arguments), the model only predicts the tactic without arguments (i.e., the core tactic). There are 49 core Coq tactics (see Section 2.2.5). Further simplification is made by grouping similar tactics together and having the model only predict the *tactic group*.

Tactic grouping has two advantages. One is to balance the dataset. Plotting the occurrence of each tactic for the proof step datasets reveals that some tactics occur far more often than others. This is shown in Figure 4.1. Unbalanced datasets are often not desirable for machine learning models. Groups are designed to "even out" the dataset, making it more balanced.

The second is that it focuses on the overall proof strategy. Usually, more than just one useful tactic can be applied in a given proof state. For instance, when the proof state is close to being proven, it often suffices to apply one of the tactics corresponding to Coq's internal small-scale inference engines (see Section 2.2.3). Which exact one is not necessarily crucial, as similar tactics solve many of the same subgoals. The grouping tries to capture the fact that it is not necessarily essential to predict which exact tactic to apply, but instead what type of tactic. Huang et al. (2019) also point out that many tactics have strong similarities.

The groups were designed based on the Coq manual (Barras et al., 1997), as well as introductory resources on Coq[2][3]. Table 4.3 details the the exact grouping, as well as the distribution of each group across the human-written proof steps.

[2]https://www.cs.cornell.edu/courses/cs3110/2017fa/a5/coq-tactics-cheatsheet.html
[3]https://coq.inria.fr/refman/proof-engine/tactics.html

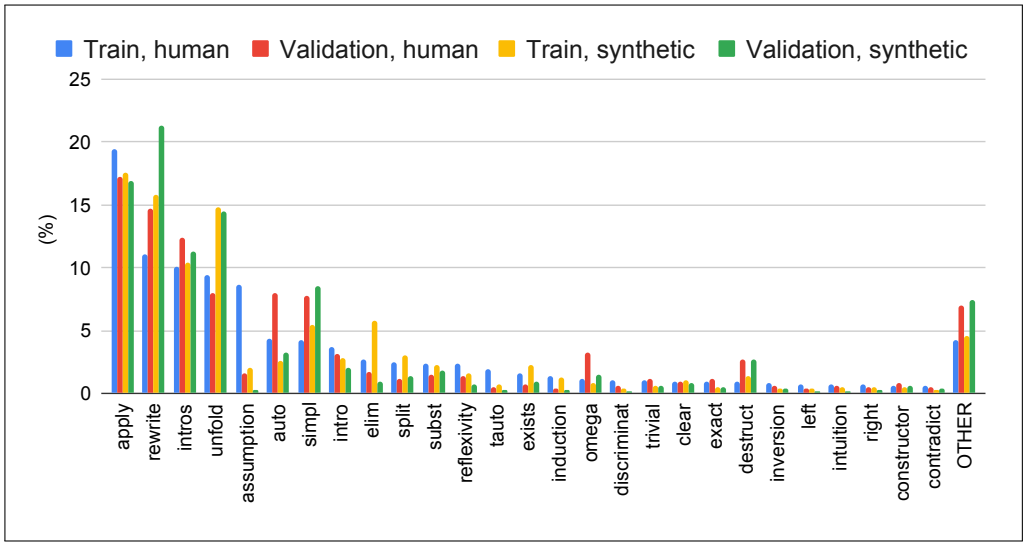


Figure 4.1: Frequency of core tactics in the proof step datasets. The least common tactics are aggregated and put in under the category OTHER.

Table 4.3: The tactic grouping.

Group	**Frequency**	**Members**
Easy goals	Train: 24.86% Validation: 21.32%	`reflexivity`, `f_equal`, `symmetry`, `assumption`, `trivial`, `easy`, `auto`, `exact`, `discriminate`, `constructor`, `contradiction`, `intuition`, `omega`, `eauto`, `tauto`, `contradict`, `ring`, `field`
Transformations	Train: 31.22% Validation: 33.66%	`intro`, `intros`, `subst`, `simpl`, `unfold`, `left`, `right`
Apply/Rewrite	Train: 30.56% Validation: 31.98%	`apply`, `rewrite`
Goal break up/Other	Train: 13.36% Validation: 13.04%	`split`, `destruct`, `inversion`, `inversion_clear`, `induction`, `elim`, `case`, `generalize`, `idtac`, `hnf`, `exists`, `red`, `congruence`, `specialize`, `clear`, `injection`, `exfalso`, `cbv`, `lia`, `cbn`, `revert`

4.3 Agent Design

In order to have an agent perform end-to-end theorem proving, one has to define the output of the models contained in the agent. This Thesis sticks with the idea that tactic application can be viewed as classification problems. However, ranking each argument

independently (i.e., a series of binary classification tasks) (Bansal et al., 2019a) will not be followed. This is due to two reasons:

1. Ranking each argument independently in every proof state is an expensive process, as noted by Paliwal et al. (2020).

2. When training the binary argument classification model, it will almost never be exposed to examples where the argument currently being evaluated was used in the given example. Therefore, there is the danger that the model will converge towards always scoring arguments with zero.

Table 4.2 showcases the later argument for CoqGym. It plots the number of tactics that use arguments from either context. Notice that most tactics use zero arguments. Furthermore, local context arguments are relatively more rare than global context arguments.

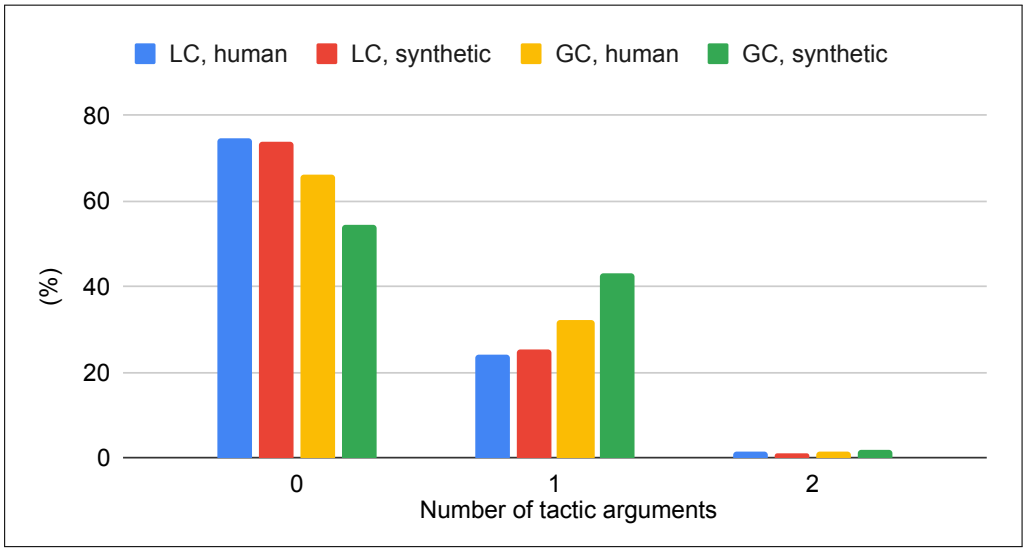


Figure 4.2: Frequency of global and local argument occurrence for the training split of the proof step datasets. GC refers to the global context and LC to the local context.

This Thesis considers tactic application instead as three distinct multi-class classification problems, each dedicated a classification model:

- C_τ: Predict which of the 49 available tactics is the most likely to be used on a given subgoal. This model only takes the current subgoal as input. The output is a probability distribution over the 49 tactics $P_{\{\tau_1,\tau_2,,\ldots,\tau_{49},\}}$.

- C_{LC}: Predict which of the n first available hypotheses in the local context is the most likely to be used on a given subgoal. This model takes the current subgoal and the n first local hypotheses as input. The output is a probability distribution over the n first local hypotheses $P_{\{h_1,h_2,\ldots,h_n\}}$.

- C_{GC}: Predict which of the m available theorems from the global context is the most likely to be used on a given subgoal. This model takes the current subgoal and the m available theorems as input. The output is a probability distribution over the m first theorems $P_{\{t_1,t_2,\ldots,t_m\}}$.

When training C_{GC}, examples not containing a theorem as an argument will be filtered out. The same will be the case for C_{LC} for local hypotheses. The models will, therefore, always have a positive example to learn from. n and m will affect the complexity of the model and the number of examples in the filtered datasets, and therefore also time and memory consumption. The higher these values are, the more tactic applications are available to the agent. In other words, there is a tradeoff between how expensive training is and the expressivity of the models.

A tactic application $\mathcal{T}$ is constructed in the simplest way possible. Whenever C_τ suggests a tactic dependent on a theorem from the global context to work, the top C_{GC} theorem is used as a tactic argument. The same is the case for tactics dependent on local context arguments to work. In this case, C_{LC} will be used to select an appropriate argument. The resulting agent is depicted in Figure 4.3. Note that, as explained in Section 2.2.2, even though ITP local context arguments can be both direct references to terms contained in subgoals and local hypotheses, the agent can still use local context argument successfully assuming it modifies local context so that subgoal terms are put among the local hypotheses. In Coq this can be done by for example the `intros` tactic (see Section 2.2.5). The "tactic building" module is independent of the classifiers, meaning it can be tailored to specific ITP systems. In this Thesis, Table 2.1 is used to ensure the agent only outputs valid tactic applications.

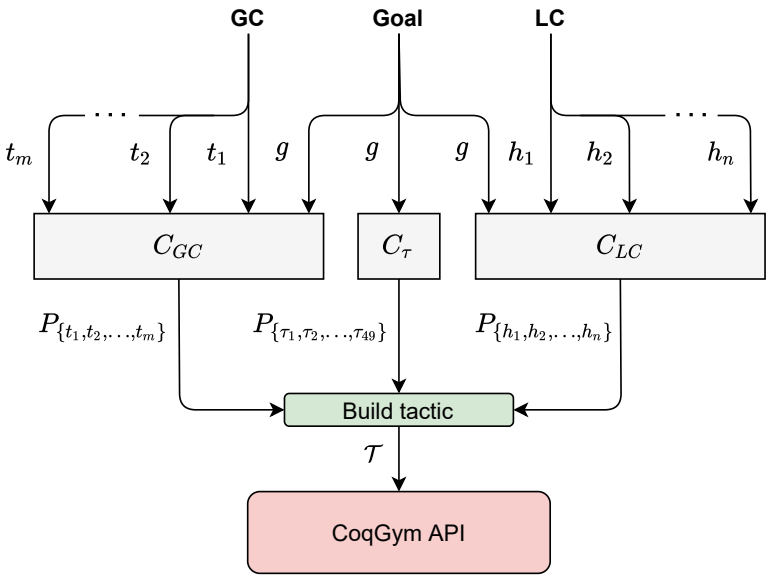


Figure 4.3: The end-to-end theorem proving agent.

4.4 Designing Architectures

This section will explain the model architectures. This means explaining different architectures for C_τ, C_{LC} and C_{GC}. Each architecture is given a name for easier reference. Hyperparameters and other concrete configurations are not included in this section. These are instead described in Section 5.1 when a concrete plan for each experiment is laid out.

All architectures follow the same overall implementation. Each classification model implements an embedding network $\mathcal{E}$ and a prediction network $\mathcal{P}$. In addition, C_{LC} and C_{GC} concatenate the embeddings corresponding to the goal and the arguments. This concatenation is then padded if there are less arguments than the set values n (for C_{LC}) and m (for C_{GC}). The resulting overall architecture is shown in Figure 4.4. Note that each classifier implements a separate embedding network.

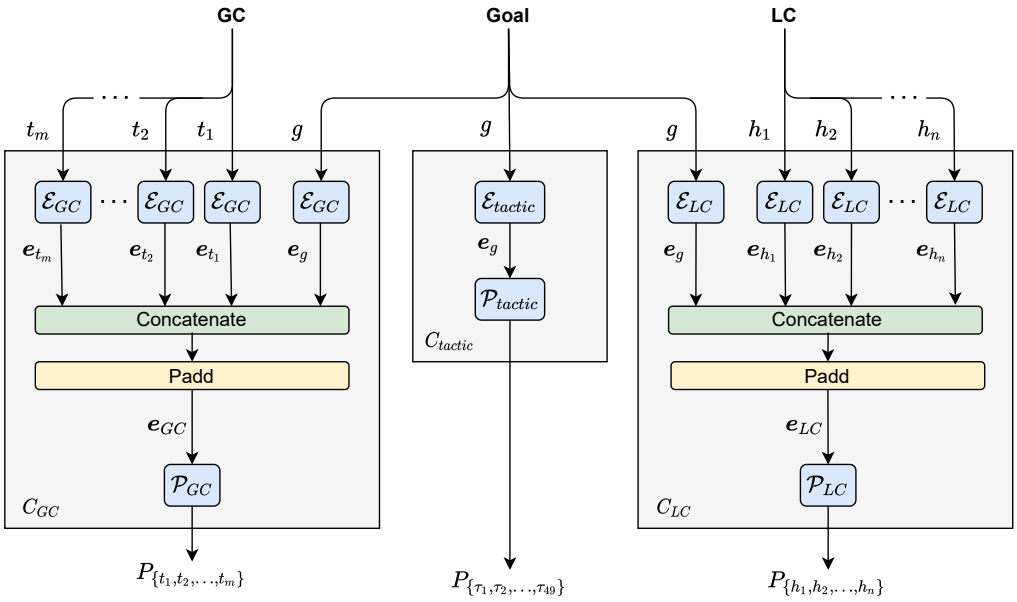


Figure 4.4: The overall end-to-end theorem proving architecture. $\boldsymbol{e}$ refers to an embedding vector, $\mathcal{E}$ to an embedding network, and $\mathcal{P}$ to a prediction network. GC refers to the global context and LC to the local context.

4.4.1 GAST – Graph Convolutional Network-based Architecture

GAST is a GCN-based architecture, inspired by Paliwal et al. (2020). The main idea is to use message passing on the nodes of the Abstract Syntax Tree (AST) representation of logical expressions. The resulting embedding is fed into the predictive model. In other words, GAST is a graph classification model.

A preprocessing step where CoqGym's Lark ASTs (see Section 3.2.4) are converted to matrix representations is needed. This is done by a simple post-order traversal of the AST, where a sparse matrix containing the one-hot encoded nodes is built X, as well as the adjacency matrix A. Crucially, the one-hot encoding is possible because CoqGym ASTs are built using values from a finite set of 55 abstract symbols – CoqGym's "nonterminals" (explained in Section 3.2.4).

Message passing can take many forms. In the tactic group experiments, a few variations are tested. A simple linear layer is used in these experiments to make predictions, and a mean operation is used as the readout function. In the end-to-end experiments, the universal graph classification architecture DGCNN from Zhang et al. (2018) is used (see Section 2.3.11 for details on DGCNN). This means that in the case of end-to-end theorem proving the GAST prediction network is a convolutional network with a dense hidden layer, and SoortPool operator from Zhang et al. (2018) is used as readout. Figure 4.5 depicts the overall model architecture of GAST.

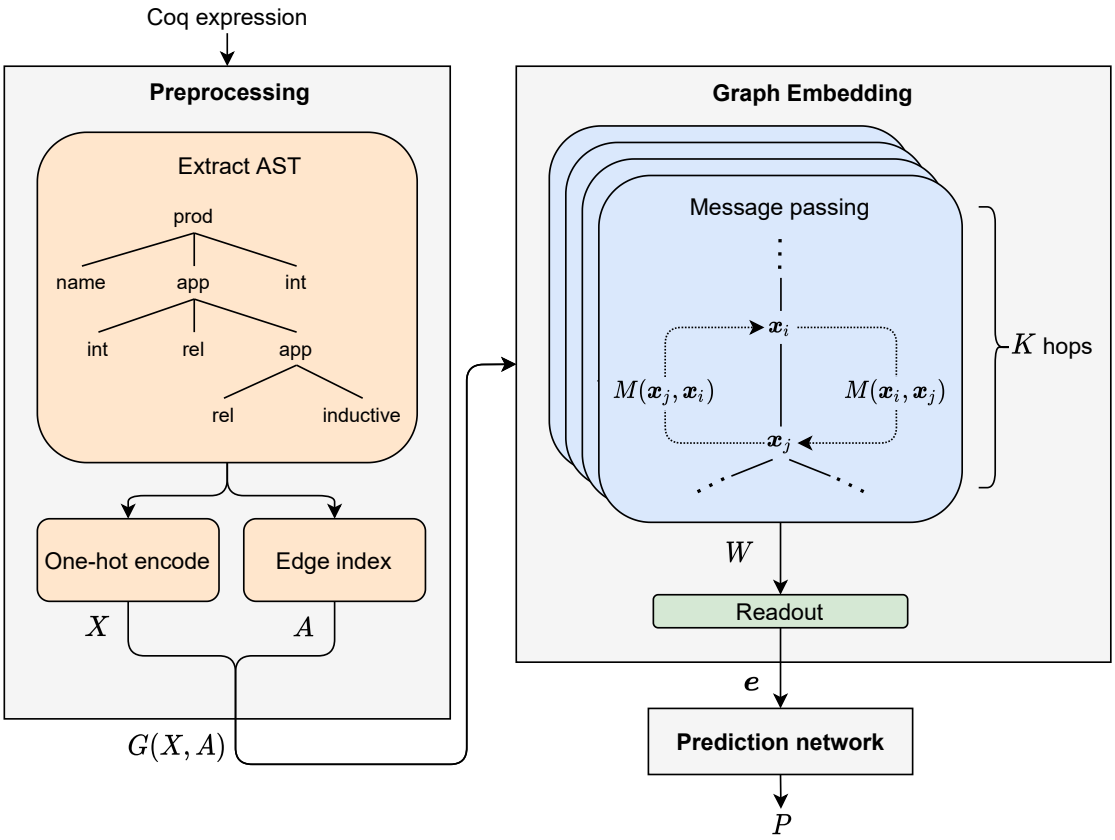


Figure 4.5: The GAST architecture. W denotes the node embedding after obtained after message passing.

4.4.2 BERTac – BERT-based Architecture

BERTac is a BERT-based (Devlin et al., 2018) architecture (see Section 2.3.12 for details on BERT). Before expressions are inputted to BERT, some preprocessing is needed. All expressions have an *identifier* which is important information to pass with the expressions. This is because the identifier is directly referenced in other related expressions. For instance, a goal expression might contain the variable H. In the local context, there can be a hypothesis attached to H. The expression of this hypothesis does not contain any reference to H, but the identifier does. In order to relate the hypothesis to the correct term in the goal expression, the identifier is needed. The input to BERT is a concatenation of (identifier, expression)-pairs. Each pair is mapped to a single sequence of the form identifier + "points to" + expression. The input sequence is tokenized using the pre-trained BERT tokenizer containing tokens for 30,522. While this works off-the-shelf, it is by no means ideal, as this tokenizer is intended for natural language and not logical expressions. The prediction network is a linear layer with a Softmax function applied to the output. Figure 4.6 shows the overall BERTac architecture.

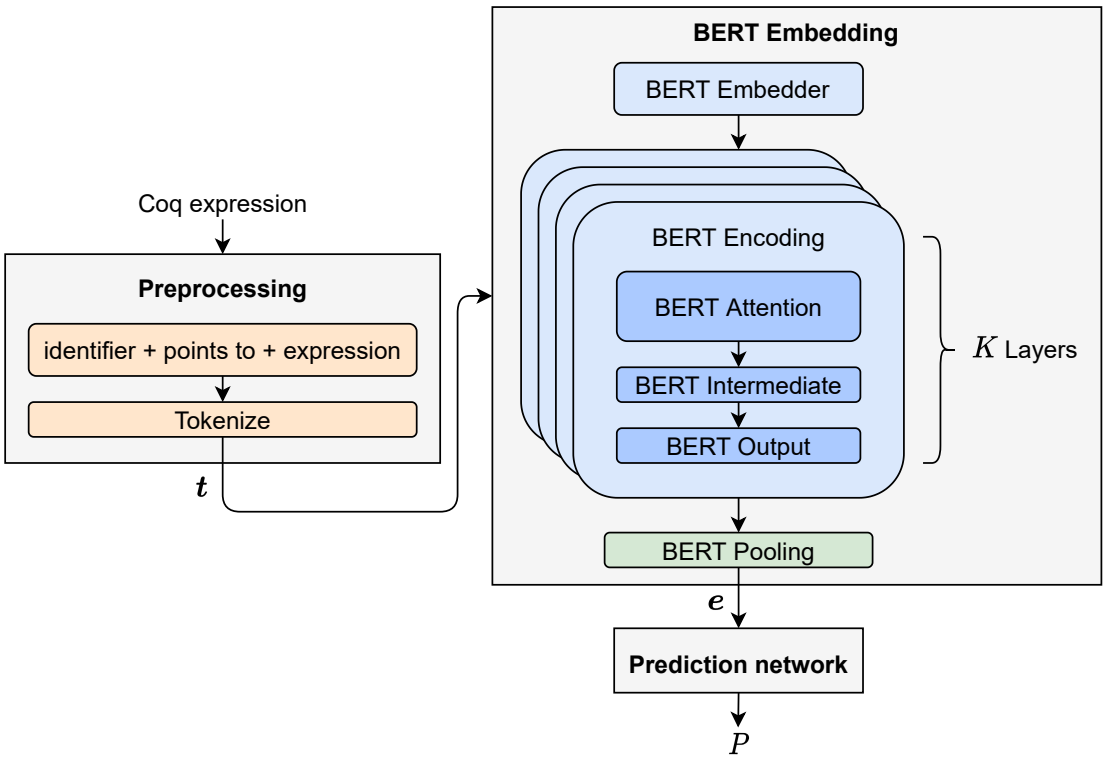


Figure 4.6: The BERTac architecture. $\boldsymbol{t}$ denotes a tokenized sequence.

4.4.3 QTac – Deep Q-learning Architecture

QTac is a deep reinforcement learning agent implementing deep Q-learning (Mnih et al., 2015) (see section 2.3.13 for an introduction to deep Q-learning). It trains by interacting with CoqGym proofs and is constructed in the same way as the agent described in Section 4.3. Furthermore, rather than having each classifier trained using deep Q-learning, only C_τ is trained this way. The Q-network implements the same model as the GAST C_τ model. C_{LC} and C_{GC} are pre-trained models, loaded from the top-performing C_{LC} and C_{GC} from the supervised learning experiments. The QTac architecture is shown in Figure 4.7.

QTac is trained using a replay memory. However, to make sure the Q-network only trains on relevant experiences, only actions that lead to a new proof state are added to the replay memory. Some actions result in an error response from Coq, even though the tactic application is valid in itself. Other actions do not respond with an error message, but the proof state is not changed (i.e., the tactic application corresponds to a loop in the proof tree). In both of these cases, the experience is discarded. A target network is used for more stable training with weights from the Q-network periodically copied to the target network. QTac will also not train on every experience in the replay memory. Instead, a mini-batch from the replay memory is picked whenever the replay memory has filled up with 20% more experiences than the mini-batch size. This process is not entirely random. Instead, experiences from successful proof attempts will be guaranteed included in the mini-batch and the remaining chosen uniformly at random. This is because the theorem proving task is difficult and QTac is likely to fail in most cases. To make sure the agent sees enough positive examples, they are guaranteed to be replayed when they occur.

In order for QTac to balance explorations and exploitation, ϵ-greedy approach is used. Furthermore, ϵ is set to decay exponentially. Two modes for training QTac are implemented:

- *Wide*: Consider each proof as an episode and decay ϵ between each proof. This means that each theorem is attempted only once. The idea is that QTac will see as many theorems a possible during training.
- *Deep*: Have QTac attempt n number of episodes for each theorem successively. ϵ is decayed between each episode and reset for each new theorem. The idea is that QTac will try to get "really good" at proving the theorems it is exposed to at the cost of seeing fewer unique theorems.

QTac can combine reinforcement learning and supervised learning by periodically training the Q-network on a labeled batch from the proof step dataset. The target network is in this case also updated after a supervised session has taken place. This is a straightforward addition as the Q-network implements the same architecture as the GAST C_τ model.

A simple reward function r is used for each new state QTac encounters, where a timeout or reaching the maximum number of tactics is considered a failed proof attempt and results in $r = -1$ (negative reinforcement). A successful proof yields $r = 1$ (positive reinforcement), and reaching a non-terminal state yields $r = 0$ (neutral reinforcement).

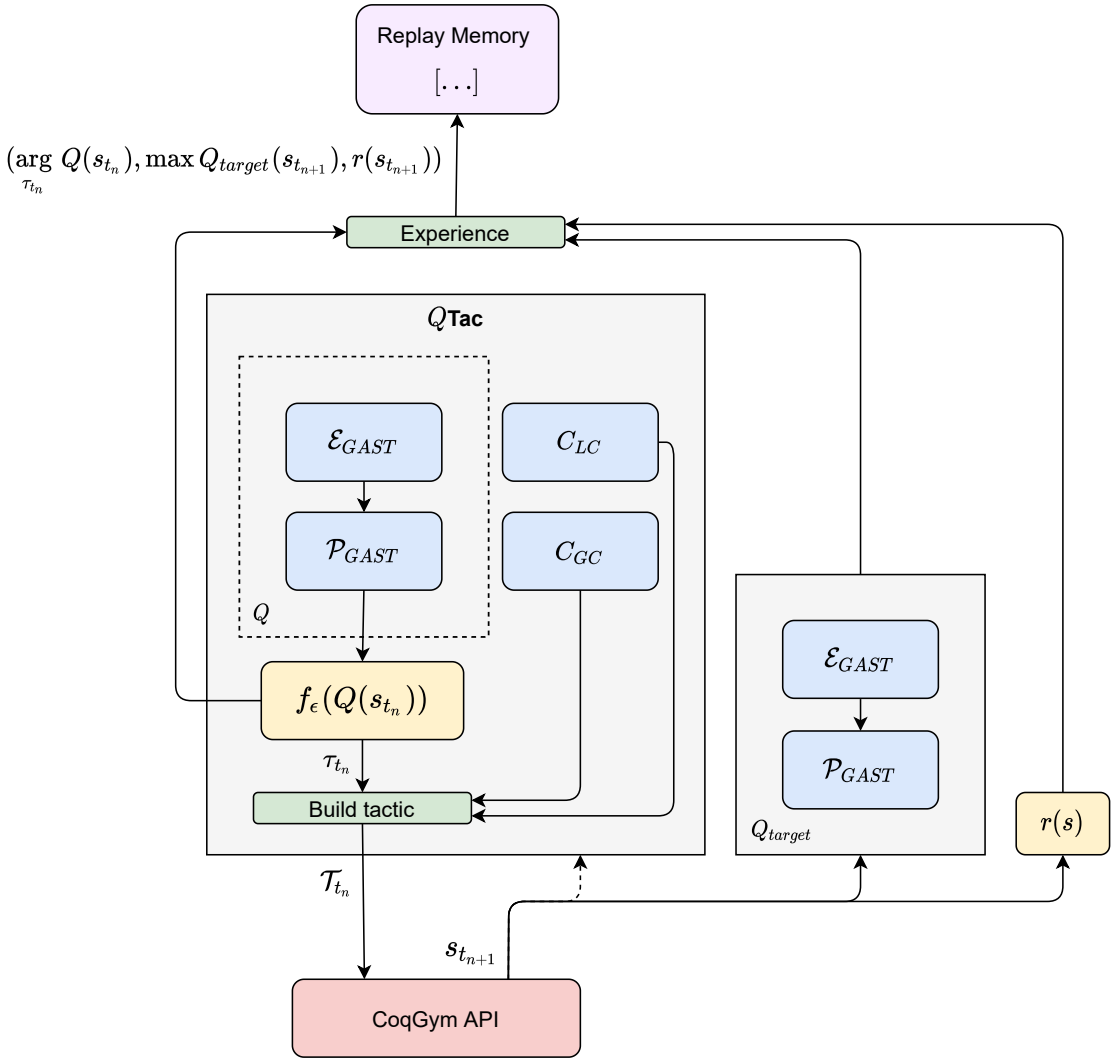


Figure 4.7: The QTac architecture. f_ϵ is the ϵ-greedy function, r is the reward function, s denotes a proof state and t_n denotes time step n.

Chapter 5

Experiments and Results

All relevant results will now be presented. This begins with an experimental plan (Section 5.1), where experiments are broken down into smaller concrete experiments and configurations are defined. Then, the experimental setup is explained (Section 5.2). This includes deep learning frameworks, setting up CoqGym, and the computational resources used to run the experiments. Finally, results are presented (Section 5.3).

5.1 Experimental Plan

Experiments in this Thesis are broken down into three categories:

- Experiment 1: Tactic group experiments.
- Experiment 2: Supervised learning models for end-to-end theorem proving.
- Experiment 3: Reinforcement learning models for end-to-end theorem proving.

Several models are trained and tested for each experiment. To keep things simple, all models are trained using the popular Adam optimizer (Kingma and Ba, 2017) (see Section 2.3.7). This is the same optimizer used in the HOList experiments (Bansal et al., 2019b,a; Paliwal et al., 2020). A mini-batch size of four will be used if not stated otherwise. Any mini-batch size higher risked running into memory issues on the available hardware resources. Cross-entropy loss will be used for all supervised models and Huber loss will be used for replay memory training as this is loss is less sensitive to outliers (see Section 2.3.4).

Three levels of regularization are defined for the experiments. These are summarized in Table 5.1, and will be referred to as *low*, *medium* and *high* regularization, for the sake of brevity. Models will be trained on one or more regularization levels, depending on indications from previous results. For an explanation of weight decay and dropout, see Section 2.3.8. Furthermore, models are *early stopped* based on validation accuracy scores computed after each training epoch.

5.1.1 Experiment 1 – Tactic Groups

The goal of the tactic group experiments is to prototype GCN-based models (the *GAST* models designed in section 4.4.1) and BERT-based models (the *BERTac* models designed

Table 5.1: The three levels of regularization defined for experiments.

	low	**medium**	**high**
Weight decay	1e-6	1e-5	1e-5
Dropout	0.1	0.5	0.7

Section 4.4.2) on labeled proof step data. The models are tasked with predicting the tactic groups defined in Section 4.2 corresponding to proof steps. Models will only be trained and validated on human-written proof steps. This assumes that good hyperparameters for models trained in human-written proofs will also be good hyperparameters for models trained on synthetic proofs. See Section 3.2.4 for an explanation of CoqGym's synthetic proof data.

Experiment 1a: Tactic Group Baselines

The tactic group proxy metric is designed in this Thesis. Therefore, there are no previous comparable results for this metric. In order to have benchmarks to compare GAST and BERTac against, the following baselines are defined:

- Weighted guesses (baseline 1).
- Most common class (baseline 2)
- Feed Forward Network (FFN) classifier (baseline 3).

Baseline 1 and 2 are straightforward. Baseline 1 makes a random guess, where the probability for picking a tactic group is weighted by the frequency of how often that tactic group occurs. Baseline 2 always guesses the most common class.

Baseline 3 is an FFN classifier. A goal encoding is passed to the input layer and propagated through two fully connected hidden layers of the same dimension as the input layer. The hidden state is then passed to a fully connected output layer of dimension four (as there are four tactic groups).

The main challenge with baseline 3 is to obtain a goal encoding. Fortunately, CoqGym provides Abstract Syntax Tree (AST) representation of Coq expressions (see Section 3.2.4). Nodes take a value from a fixed-size space. An encoding for baseline 3 is obtained by simply counting the number of each nonterminal in the goal AST. Note that this means the relational semantics in the relationship (i.e., the AST edges) are lost.

There are 55 nonterminals, meaning this will be the dimension of the input and hidden layers. Baseline 3 therefore consists of $55 \cdot 55 \cdot 2 + 55 \cdot 4 = 6{,}270$ parameters in total. Dropout and ReLU activation is used between each layer before Softmax is applied on the output logits to obtain a probability distribution over the four tactic groups. The model will be trained on both low and medium regularization levels.

Experiment 1b: GAST on Tactic Groups

A variety of hyperparameters and message passing algorithms will be tested for the GAST model. Two phases are defined for this experiment:

1. Phase 1: Vary message passing algorithm. Fix everything else.
2. Phase 2: Vary the complexity and regularization of the network. Fix everything else.

Although everything except message passing stays fixed in phase 1, models will still run on low and medium regularization levels. This is to help determine what regularization level is best suited for GAST. Three message passing algorithms will be tested in phase 1:

- Multilayer Perceptron (MLP). A custom, and simple, message passing algorithm based on the message passing algorithm from Paliwal et al. (2020). The encodings of two adjacent nodes are passed through an FFN with one hidden layer to compute node embeddings.
- Graph Convolutional Network (GCN). Message passer from Kipf and Welling (2017) (see Section 2.3.11).
- Simple Graph Convolutions (SGC). Message passer from Wu et al. (2019) (see Section 2.3.11).

MLP is included to see if an MLP message passing technique can compete with the more sophisticated convolution-based techniques. It is also similar to the implementation in Paliwal et al. (2020), which showed improvements in the HOList framework. GCN (Kipf and Welling, 2017) is arguably the most adopted message passing technique and serves as a natural starting point for GCN implementations. SGC (Wu et al., 2019) is interesting because it is essentially the same algorithm as GCN, only simplified. This makes it a faster algorithm while still being competitive with GCN (Wu et al., 2019). Models are trained and validated for eight epochs in this experiment. As a comparison, Yang and Deng (2019) train ASTactic for four epochs.

Some hyperparameters are not focused on and remain fixed. A linear layer serves as the prediction network, and only a single round of message passing (the number of *hops*, see Section 2.3.11) will be used. The learning rate will simply be set to 1e-3. This is higher than a learning rate of 3e-5, used by Yang and Deng (2019) and First et al. (2020), and 1e-4, used by Paliwal et al. (2020). A node embedding of 256 is chosen as default – the same as ASTactic (Yang and Deng, 2019) and TacTok (First et al., 2020). A simple mean operation will be used as the readout function, to globally pool node embeddings to a fixed size graph embedding of size 256. ReLU activation is used between each neural network layer.

Only SGC will be used during phase 2, as this is the fastest to run. These

experiments will be run for 20 epochs instead of just eight, as this is not too computationally expensive when using SGC. It will also allow the models a better chance of escaping local minimums. Three configurations will be tested. These are summarized in Table 5.2. The main goal is to compare low and medium regularization further and see if adding complexity to the network increases performance.

Table 5.2: GAST configurations for phase 2 of experiment 1b.

	default	**reg.**	**complex**
Node emb. dim.	256	256	1024
Message passing	SGC	SGC	SGC
Readout	Mean	Mean	Mean
Prediction network	Linear	Linear	Linear
Hops	1	1	4
Regularization	low	medium	medium

Experiment 1c: BERTac on Tactic Groups

The focus for experiment 1c is not the BERT architecture, but rather regularization levels and learning rate. Three configurations will be tested for BERTac. These are summarized in Table 5.3. Note that fixed BERT-specific configurations are also included in the table.

Table 5.3: BERTac configurations for experiment 1c.

	low reg.	**low α**	**medium reg. + low α**
Regularization	low	low	medium
Learning rate (α)	1e-3	1e-6	1e-3
Tokenizer length	512	512	512
Vocabulary size	30,522	30,522	30,522
Hidden layers	6	6	6
Attention heads	6	6	6

The drop in learning rate is due to recommendations provided by Devlin et al. (2018). They suggest that a learning rate in the range 1e-5 is usually preferable for BERT.

Tokenizer length and vocabulary size are simply set to the default values used in the original BERT implementation. The number of hidden layers is reduced from 12 to 6. The same is done for the number of attention heads. This is to speed up the training process. While it would be interesting to drill deeper into the BERT architecture, time constraints dictated that these experiments stayed reasonably off-the-shelf.

5.1.2 Experiment 2 – Supervised Learning

Table 5.4 summarizes the configurations used in experiment 2. For both GAST and BERTac, a tactic classifier C_τ, a local context classifier C_{LC} and a global context classifier C_{GC} needs to be trained (see the theorem proving agent described in Section 4.3). Three different models will be trained for each classifier – one on human-written proofs, one on synthetic proofs, and one on both datasets.

Table 5.4: Configurations for experiment 2. LC denotes the local context and GC the global context.

Datasets	human, synthetic, both
n **(hypotheses from LC)**	10
m **(theorems from GC)**	10
d **(depth limit)**	10, 50, 100
k **(beam width)**	5, 10, 20
Regularization	medium, high

Results from experiment 1 indicated that models should at least be trained using medium regularization. The models will therefore use this setting as a default. A version using high regularization will also be tested to see the effect of increased dropout (see Table 5.1). Only the models with the highest validation scores will be used to build theorem proving agents.

When designing the theorem proving agent the variables n and m were defined (see Section 4.3). These correspond to the number of hypotheses to include from the local context (n) and the number of theorems to include from the global context (m). m is simply chosen to be the same as Yang and Deng (2019) and First et al. (2020): $m = 10$. To decide a reasonable value for n, observe Table 5.1. This table plots the percentage of the proof step datasets with n number of hypotheses in the local context. Most proof steps have a local context consisting of less than 20 hypotheses. Setting $n = 20$ therefore seems like a reasonable choice. Unfortunately, the ASTs can be fairly large, and GAST needs to deal with one more AST for every added hypothesis. That is, increasing n by one corresponds to one more AST. Therefore, $n = 10$ will be used instead of $n = 20$. This decreases the complexity of C_{LC} models, which is particularly important for GAST C_{LC} models as too many ASTs can result in memory issues. The downside is that C_{LC} will be able to deal with fewer proof states than if n is was higher.

Recall too that both C_{LC} and C_{GC} are only trained on examples where a true correct argument exists (explained in Section 4.3). This means that lowering n and m decreases the dataset sizes C_{LC} and C_{GC} train on. The resulting dataset sizes for the classification models, when $n = 10$ and $m = 10$, are shown in Table 5.5. As can be seen in the table, there is a significant drop in the dataset sizes for the argument models.

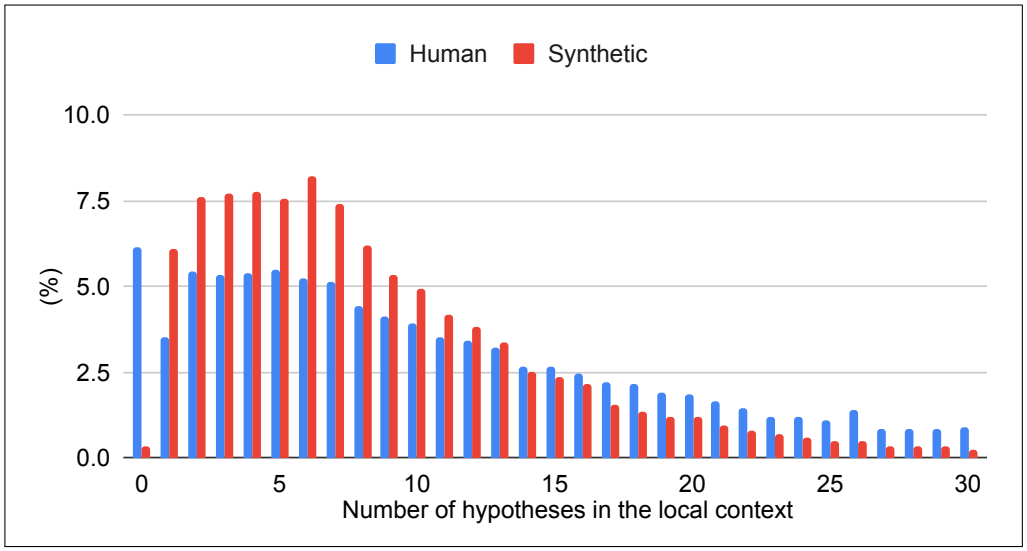


Figure 5.1: The percentage of proof steps that have n number of hypotheses in the local context.

Table 5.5: The dataset sizes for the supervised learning models. The numbers are presented in the format "train / validation".

	Human	**Synthetic**	**Both**
C_τ	121,764	174,076	295,720
	68,180	113,048	181,228
C_{LC}	16,492	30,350	46,842
	9,705	19,862	29,567
C_{GC}	6,108	14,464	20,572
	3,280	8,688	11,964

Depth-First Search will be used to traverse the proof tree, same as ASTactic and TacTok. In addition, a beam width of the top k tactic applications is calculated for each proof state so that the agent can apply a new tactic whenever it has to backtrack the proof tree. A depth limit d is used to limit how far down a branch in the proof tree the agent traverses. As default values $d = 50$ and $k = 10$ will be used. d is chosen to be the same value as Yang and Deng (2019). Although $k = 10$ is not the optimal value for TacTok or ASTactic, it is chosen as default because it speeds up the test process (Yang and Deng, 2019). Note that experiment 2e will address different values for d and k (including the optimal $k = 20$ from Yang et al. (2016); First et al. (2020)), to see what impact these variables have on performance.

The default timeout and the total number of tactics the agent is allowed to try

before giving up on a proof search is set to the same as experiments in Yang and Deng (2019) and First et al. (2020). This is a timeout of ten minutes and a limit of 300 tactics for each theorem.

Experiment 2a: Random Guessing Baseline

A random guessing agent, using the design from Section 4.3 will be tested as a baseline. Because this baseline implements the same agent design as the rest of the agents in this Thesis, results can be compared directly. Whenever either a global context or local context dependent tactic is guessed, a random guess over the corresponding argument space will be made. The random guessing agent will be tested with the default $d = 50$ and $k = 10$ values, and again with updated d and k values if experiment 1e shows that different values improve results.

Experiment 2b: Supervised GAST Models

For this experiment, the learning rate will be set to 1e-3 and node embedding size to 256, based on results from tactic group experiments. The readout function in this architecture is the SortPoll operator from Zhang et al. (2018) that selects the top k features from the node features. The top $k = 30$ features will be pooled.

Experiment 2c: Supervised BERTac Models

For this experiment, the learning rate will set to 1e-6, based on results from tactic group experiments. The number of hidden layers and attention heads is reduced from 12 to 6 to speed up the training process.

BERTac will also be trained and validated in one additional setting: loading pre-trained BERT model weights before training. Although it seems unlikely that weights obtained from classical NLP pre-training tasks will help a model trying to reason about logical expressions, it is included out of curiosity.

Experiment 2d: Combining GAST and BERTac Models

It is possible to combine the classifiers in arbitrary ways, as each classifier making up the theorem proving agent is independent of each other. Suppose, for example, a GAST model is the best core tactic classifier (C_τ), and BERTac models are the best argument classifiers (C_{LC} and C_{GC}). In that case, it is trivial to have the agent load the GAST model as its tactic classifier and the BERTac models as its argument classifiers. For each proof step dataset, the best performing classifier from each architecture will be combined to form a "best" agent.

Experiment 2e: Changing depth limit and beam width

In order to see if the depth limit d and beam width (i.e., the number of tactic candidates) k impact the performance, a few variations of these values will be tested. d will be set to 10 and 100, and k to 5 and 20. If the result is better for different d or k than the default $d = 50$ and $k = 10$, the best combination will be tested. This experiment will only use the overall best-performing end-to-end agent from previous experiments. Yang and Deng (2019) also tests ASTactic with different k values. They found that $k = 20$ is optimal. However, they only test with $d = 50$. First et al. (2020) use $k = 20$ and $d = 5$.

5.1.3 Experiment 3 – Reinforcement Learning

Since the deep Q-learning agent QTac (described in Section 4.4.3) trains by interactive proof attempts, it is subject to the SerAPI bottleneck (see Section 3.2.4). This means that exposing QTac to the whole training dataset within a reasonable time frame can be challenging. Yang and Deng (2019) provide the average time used to prove theorems for ASTactic. This is 2.2 seconds when $k = 10$. With this in mind, a time limit of only three seconds, as opposed to the default 10 minutes, will be used when training QTac. This drastically lowers the time spent on each proof; QTac will see more proofs in a shorter amount of time, at the cost of failing on proofs it potentially could have solved given more time. This is not a major issue if one assumes that most proofs are solved within three seconds.

QTac's Q-network follows the same architecture as the GAST C_τ model. This means that it is possible to train the QTac agent with labeled examples, in addition to replay memory training. Bansal et al. (2019b) use this approach when training their reinforcement agent in HOList. To help QTac in the training phase, the same idea will be used here. However, instead of only using an initial supervised learning phase before reinforcement learning is deployed, as done by Bansal et al. (2019a), supervised training will be interleaved with the reinforcement learning process. Specifically, after each 1,000 proof attempt, QTac will be supervised on 2,000 synthetic proof steps. This means that an imitation proof style will influence QTac less in the highly explorative beginning phase of training. Hopefully, this increases the chance that QTac finds its own "style" of proving theorems, which is the main goal for this agent. The mini-batch size is simply set to one during the supervised training.

Regularization techniques are less common in reinforcement learning than in supervised learning. This usually is fine when the test task is identical to the training task (e.g., when teaching an agent to play chess), as overwitting is not a concern. However, in the case of theorem proving, each proof is different and the agent must learn a general approach to the proof procedure. Regularization can sometimes help increase generalization. Therefore, two approaches will be tested: (1) use low regularization while training, and (2) use medium regularization. Regularization will apply to both the Q-network and the target Q-network.

When training the Q-network in a supervised fashion, the learning rate will be set to 1e-3 (the same as for the supervised learning models). A lower learning rate of 1e-5 will be used for replay memory training. The reason for this is that replay memory includes a considerable amount of noisy training data. The theorem proving task is hard, and QTac will most likely fail in most cases. It is not desirable that QTac steps too far along gradients when seeing potentially highly irrelevant experiences.

Interacting via SerAPI remains a bottleneck even with a time limit of three seconds. This is because SerAPI sometimes has to wait several minutes before Coq responds to a tactic application, making the three-second timeout . QTac will, therefore, only be trained on 10,000 theorems. This is less than 25% of the full training set in CoqGym. QTac will do supervised learning on $10 \cdot 2,000 = 20,000$ proof steps. Setting the maximum number of tactic applications for each proof attempt to 50 means each proof attempt will typically generate around 50 replay experiences, as QTac is expected to fail on most theorems (i.e., by using up all 50 tactic applications). The resulting experiences QTac can potentially train on amounts to $10,000 \cdot 50 = 500,000$. As mentioned in Section 4.4.3, not all replay memories are used for training. Instead, a random sample of 256 is pooled whenever the replay memory exceeds 307 experiences (20% more than the replay batch size of 256). This means that QTac will train on around 400,000 replay experiences from 10,000 proof attempts.

Experiment 3a: Wide QTac

As mentioned in Section 4.4.3, two training modes will be used for QTac. One is the *wide* mode. In this mode, QTac only sees each theorem once. The exploration rate ϵ is decayed after each proof attempt. ϵ will start at 1.0 and approach 0.2. The decay rate is set to 3e3 in this mode. When trained on 10,000 proofs, this means ϵ will be ~0.2 at the end of the training session.

Experiment 3b: Deep QTac

QTac is also trained in the *deep* mode. The idea is that QTac will attempt each theorem 10 times before moving on to the next.

However, CoqGym does not support a straightforward way to handle individual theorems. Instead, the agents interact with *proof files*. This does not matter for most cases (and has therefore not been mentioned before now), but when deciding wide QTac's ϵ decay, it does. Each proof file contains an unknown, small number of theorems. Therefore, a simple solution will be used to *almost* train QTac in the ideal deep mode. Namely, have QTac train on the same proof file 10 times before moving on to the next. It is simply assumed that each file contains *around* 20 theorems each.

ϵ will be decayed so that it starts at 1.0 and ends at 0.2 when $10 \cdot 20 = 200$

proof attempts are reached. ϵ decay is set to 3e1, meaning that ϵ ends at ~0.2 when reaching 200 theorems. ϵ is then reset. In this mode, QTac will only be exposed to $\frac{10{,}000}{10}$ = 1,000 *unique* proofs, as each proof is attempted 10 times.

5.2 Experimental Setup

This section explains the different frameworks and resources used in the experiment implementation. The code for all end-to-end experiments is available in a CoqGym fork on GitHub[1]. The code for the tactic group experiments is available in a separate repository[2].

5.2.1 Deep Learning Frameworks

All models are implemented using PyTorch[3]. PyTorch is a popular deep learning framework which is high-level, flexible and allows for easy integration with Nvidia GPUs through the CUDA API.

PyTorch Geometric[4] (Fey and Lenssen, 2019) is used to implement GNN models. This framework is a general-purpose GNN framework built on top of PyTorch. It provides easy implementations of both custom and pre-implemented message passing algorithms[5]. The framework also implements additional graph-related functionality, like readout functions and graph batch handling.

The Hugging Face[6] implementation of BERT is used. Hugging Face provides an API to extract both general-purpose Transformer implementations and specific implementations such as BERT. Furthermore, one can load in pre-trained versions of both a BERT tokenizer and the BERT model itself, or simply the architecture with no specific weight initialization. The API is very flexible and lets the user specify BERT details, like the number of hidden layers, the number of attention layers, and the number of unique tokens in the tokenizer.

5.2.2 CoqGym Setup

A handful of libraries are needed when using CoqGym. A specific version of OCaml[7] must be set up with the OPAM package manager[8]. OCaml is the functional programming language in which Coq is written. The Coq projects comprising the CoqGym proof

[1] https://github.com/MaganMK/CoqGym
[2] https://github.com/MaganMK/prox
[3] https://pytorch.org/
[4] https://pytorch-geometric.readthedocs.io/en/latest/
[5] https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#convolutional-layers
[6] https://huggingface.co/
[7] https://ocaml.org/
[8] https://opam.ocaml.org/

data have to be built, a process taking around four hours. CoqGym's caching system also leverages a library called Lightning Memory-Mapped Database (LMDB)[9], which needs to be available for CoqGym to work properly. Some Coq projects also use Ruby, meaning that Ruby also has to be installed for the full dataset to be available. A detailed description of setting up CoqGym can be found in the README on the official GitHub repository[10].

When building the various Coq projects, an issue occurred for the `coquelicot` project. Coq claims that this project makes inconsistent assumptions over another Coq library `ssreflect`. Despite continued efforts, the problem was never resolved. This means that the `coquelicot` project remained broken and could not be used for experiments in this Thesis. `coquelicot` is part of the CoqGym test set, meaning that agents are not evaluated on this Coq project, reducing the test set from 13,137 theorems to 11,670 theorems.

5.2.3 Computing Resources

All experiments were run using the NTNU Idun HPC cluster[11] (Själander et al., 2019). Idun provides both high-end CPU and GPU clusters. Experiments were primarily run on Tesla P100 GPU clusters with four to eight logical cores. The exception was end-to-end theorem proving experiments as matrix computations were not the main bottleneck but rather the CoqGym SerAPI calls. These experiments were run on CPU clusters with Intel Xeon cores. The tactic group experiments took between one and three days to run, depending on the number of epochs. Training supervised learning models for the end-to-end theorem proving task took around two days. Training reinforcement learning agents took around two days as well, when proof search timeout was set to three seconds. Testing models on end-to-end theorem proving took around three days, with a ten minute time limit for each proof.

5.3 Experimental Results

All relevant results will now be presented. An overview of best-performing theorem proving agents is provided in Table 5.6. These are the main results from experiments 2 and 3. Using the default depth limit d of 50 and beam width k of 10, the baseline random guesser can prove 6.87% of CoqGym's test set. Keeping $d = 50$ and $k = 10$, the best-performing supervised learning agent proves 8.65% of the theorems. This is 25.91% more theorems than the random guessing baseline. Note that ASTactic is also included in the table as the main comparison for this model, as it is tested using the same values for d and k. The best-performing supervised agent with $d = 50$ and $k = 10$ scores 2.15 percentage points lower than the corresponding ASTactic model.

[9] https://symas.com/lmdb/
[10] https://github.com/princeton-vl/CoqGym
[11] https://www.hpc.ntnu.no/idun

When modifying d and k, a score of 9.98% is achieved for the supervised agent. The random baseline also slightly improves results to 7.27%, when the same d and k values are used. This means that the best-performing supervised learning agent proves 37.28% more theorems than the corresponding random guessing agent and 2.92 percentage points lower than the state-of-the-art TacTok model (First et al., 2020).

The best-performing wide QTac agent proves 10.63%, and the best-performing deep QTac agent proves 10.74%. 10.74% is the highest score for any agent in this Thesis, ending up at 47.73% more theorems proved than the corresponding random guessing agent, and 2.16 percentage points lower than TacTok.

Table 5.6: Main end-to-end theorem proving results from experiments 2 and 3. `h` indicate the model was trained on human-written proof steps and `s` that it was trained on synthetic proof steps. `G` indicates that the model was a GAST model, and `B` that it was a BERTac model. Only the best-performing combination of models is included in this overview. The `easy` baseline corresponds to the best-performing Coq internal automatic engine.

Agent	C_τ	C_{LC}	C_{GC}	d	k	**Test accuracy**
`easy` (baseline 1)	-	-	-	-	-	4.90%
random guesser (baseline 2)	-	-	-	50	10	6.87%
random guesser (baseline 2)	-	-	-	10	20	7.27%
ASTactic	-	-	-	50	10	10.80%
ASTactic	-	-	-	50	20	12.20%
TacTok (state-of-the-art)	-	-	-	5	20	12.90%
BERTac	`B`, `s`	`B`, `h`	`B`, `s`	50	10	7.99%
GAST	`G`, `s`	`G`, `h`	`G`, `s`	50	10	8.65%
GAST	`G`, `s`	`G`, `h`	`G`, `s`	10	20	9.98%
QTac, wide	-	`G`, `h`	`G`, `s`	10	20	10.63%
QTac, deep	-	`G`, `h`	`G`, `s`	10	20	10.74%

5.3.1 Results from Experiment 1

The main results from experiment 1 are shown in Table 5.7. Both GAST and BERTac beat the FFN benchmark. However, this is only by a few percentage points. Furthermore, BERTac performed slightly better than GAST. All deep learning models significantly outperform the weighted random guesser (baseline 1) and the most common class (baseline 2).

Table 5.7: Main results from experiment 1.

Model	Validation accuracy
Weighted guesses (baseline 1)	27.80%
Most common class (baseline 2)	33.66%
FFN (baseline 3)	48.86%
GAST	51.28%
BERTac	**52.58%**

Results from Experiment 1a: Tactic Group Baselines

Baselines 1 and 2 are computed based on the validation set statistics and achieve accuracies of 27.80% and 33.66%, respectively.

Baseline 3 (the FFN model) is trained for 30 epochs. Figure 5.2 plots the validation accuracy of both the low and medium regularized models. The model does better when medium regularization is applied, with the best accuracy of **48.86%**.

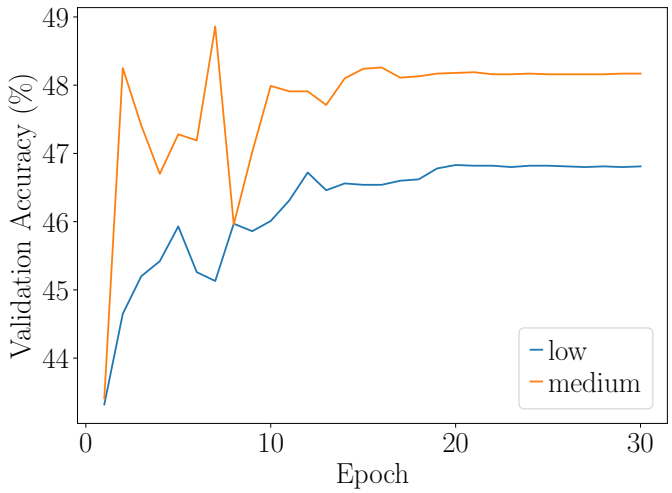


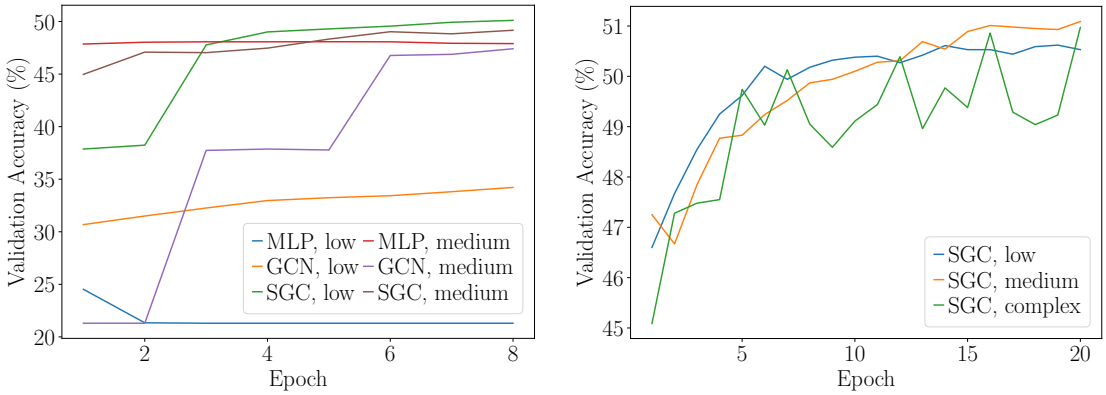
Figure 5.2: Validation accuracy plots for FFN baseline from experiment 1. "low" and "medium" refer to regularization levels.

Results from Experiment 1b: GAST on Tactic Groups

Figure 5.3a plots the validation accuracy from phase 1 of experiment 1b. These are the GAST models tested with different message passing algorithms. Both the MLP and GCN models benefit from increased regularization. This is consistent with the FFN baseline. SGC performs well with both low and medium regularization. This might be because SGC implements a less complex message passing algorithm (see Section 2.3.11),

and regularization is typically needed for more complex networks (see Section 2.3.8). All models seem to converge towards an optimum at around 50% accuracy.

Figure 5.3b plots the validation accuracy from models in phase 2 of experiment 1b. GAST seems to benefit from training for longer than eight epochs as the models approach 52% when trained for 20 epochs. Medium regularization helps GAST even though SGC is the implemented message passing algorithm. The benefit from using medium regularization can be seen from epoch 14 onward. Increasing node embedding and the number of hops (the "complex" model) results in a more unstable learning process and does not result in a better validation score. The best performing GAST model achieves a validation score of **51.28%** when medium regularization is applied, SGC is the message passing algorithm, and the model is allowed to train for 20 epochs.



(a) GAST using different message passing techniques.

(b) GAST using different levels of regularization and network complexity.

Figure 5.3: Validation accuracy plots for GAST models from experiment 1. "low" and "medium" refer to regularization levels.

Results from Experiment 1c: BERTac on Tactic Groups

The validation accuracy plots from experiment 1c are shown in Figure 5.4. Even with much less tuning than GAST, BERTac performs slightly better. The best score is achieved by the low α model, at **52.58%** accuracy. This is consistent with suggestions from (Devlin et al., 2018), where a learning rate around 1e-5 is recommended.

Comparing Figure 5.4 with Figure 5.3b shows how BERTac converges faster towards an optimum than GAST. Increasing regularization does not seem to have any impact on performance for BERTac. It hinders the convergence somewhat during the first few epochs.

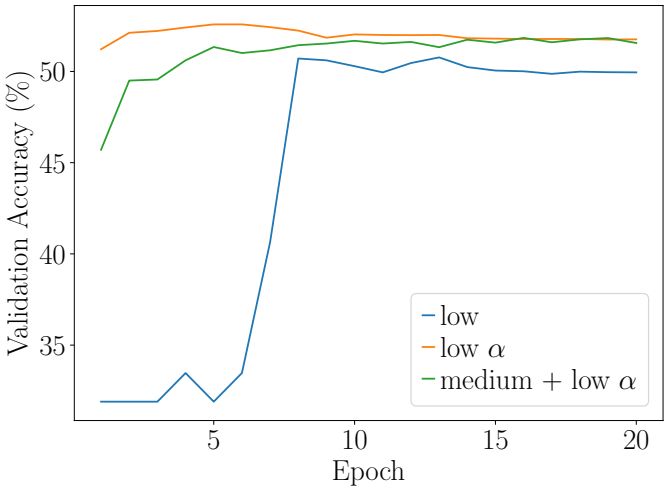


Figure 5.4: Results from of experiment 1c. BERTac is tested with different levels of regularization and learning rates. "low α" means that learning rate is reduced from 1e-3 to 1e-6. "low" and "medium" refer to regularization levels.

5.3.2 Results from Experiment 2

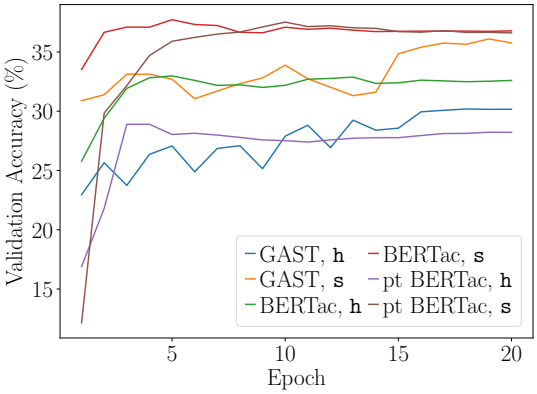
Validation Accuracy for C_τ, C_{LC}, and C_{GC} Models

The validation scores for each of the three classifiers, for both GAST and BERTac, are shown in Table 5.8. The validation scores for BERTac with pre-trained weights are included as well. BERTac scores the highest on tactic classification but significantly lower on both argument classification tasks. Both C_τ and C_{GC} experience performance gains when trained (and validated) on synthetic proof steps rather than human-written proof steps. However, C_{LC} performance is higher when trained (and validated) on human-written proof steps. Training on both human-written and synthetic proof data does not help any of the models. This points to differences in the human-written and synthetic proof data, making it counterproductive to learn from both datasets. It does not help BERTac to load pre-trained BERT weights, meaning classic NLP-style upstream training does not transfer well to formal expressions in this setting.

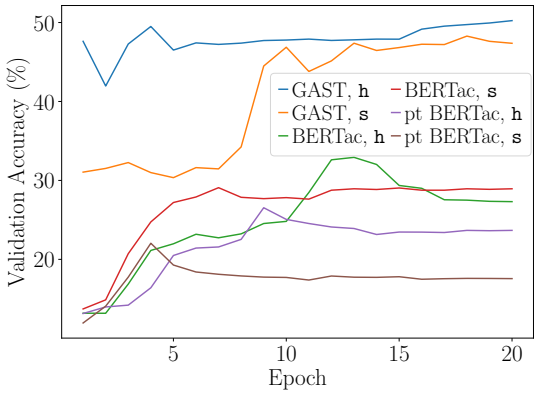
Validation plots are shown in Figure 5.5. Figure 5.5a shows how GAST and BERTac perform similarly on the tactic classification task when trained on synthetic proofs. Differences are bigger when training on human-written proofs. Figure 5.5b and 5.5c show how all GAST models outperform BERTac models on argument classification tasks. The plots also show that the type of training data impacts performance for all classifiers. Plot 5.5d is included to showcase how GAST performs better when high regularization is applied, while BERTac performs better when medium regularization is applied. This is consistent with experiment 1, where increasing regularization from low to medium improved results for GAST but slightly decreased results for BERTac.

Table 5.8: Validation accuracy for GAST and BERTac C models.

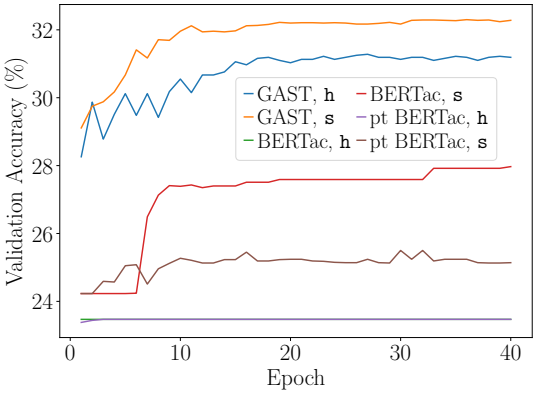
Model	Dataset	GAST	BERTac	Pre-trained BERTac
C_τ	human	30.19%	32.98%	28.90%
	synthetic	36.10%	**37.72%**	37.52%
	both	34.46%	36.14%	-
C_{LC}	human	**50.24%**	32.92%	26.54%
	synthetic	48.28%	29.08%	22.04%
	both	45.44%	32.56%	-
C_{GC}	human	31.28%	23.47%	23.47%
	synthetic	**32.30%**	27.97%	25.50%
	both	31.08%	27.06%	-



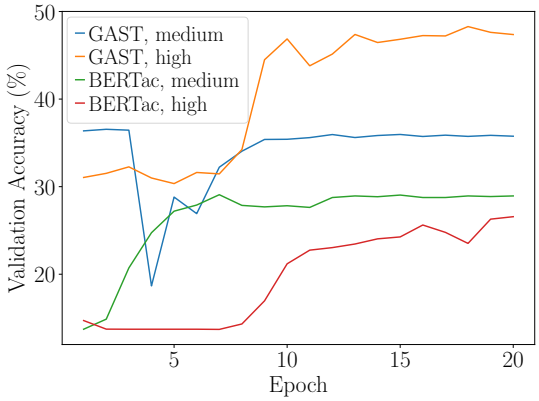
(a) C_τ models from experiment 2.



(b) C_{LC} models from experiment 2.



(c) C_{GC} models from experiment 2.



(d) C_{LC} models with different regularization.

Figure 5.5: Validation accuracy plots for C models. "pt" denotes "pre-trained". `h` refers human-written proof steps and `s` to synthetic proof steps as the training data.

End-to-End Theorem Proving Accuracy

The end-to-end theorem proving accuracy for each agent from experiment 2b-d is shown in Table 5.9. These are agents combining different classifiers based on Table 5.8. All agents benefit from training on synthetic proofs over human-written or both datasets. Moreover, performance for GAST and BERTac agents is improved further when having C_τ and C_{GC} trained on synthetic proofs and C_{LC} trained on human-written proofs. This is consistent with the validation scores from Table 5.8.

However, it is not necessarily the case that it is best to use BERTac as the C_τ model and GAST as the argument models. This is somewhat inconsistent with the validation scores, as validation scores for BERTac C_τ models are higher than for GAST C_τ models, regardless of the dataset. The best agent consists of only GAST models, where C_τ and C_{GC} are trained on synthetic proofs, and C_{LC} is trained on human-written proofs. This agent proves 8.65% of the test set.

Table 5.9: Performance of GAST and BERTac on end-to-end theorem proving. Each column indicates what proof step data the models were trained on. "Best dataset" refers to each classifier being trained on the best proof step data for that specific classifier (best meaning the highest validation score). "Best combination" combines the best GAST and BERTac classifiers, based on validation scores.

	Human	**Synthetic**	**Both**	**Best dataset**
GAST	7.92%	8.46%	8.29%	**8.65%**
BERTac	6.39%	7.71%	7.75%	7.99%
Best combination	7.04%	8.62%	8.36%	8.61%

For experiment 2e, only the best performing agent from experiment 2b-d is used: the "best dataset" GAST agent (see Table 5.9). Results for different depth limits d and beam widths k are shown in Table 5.10. Results for different d values show that lowering d to 10 improves results. This indicates that the agent should focus on shorter proofs, searching wider in the proof tree rather than deeper. Increasing k to 20 improves performance, and decreasing it to 5 significantly lowers performance. This is consistent with results from (Yang and Deng, 2019) – ASTactic proves 6.5%, 10.8%, and 12.2% for beam widths 5, 10, and 20, respectively. It is easier for the agent to perform well if it also evaluates lower probability tactic candidates in proof states. Note that the maximum number of tactic applications allowed in a proof search is always the same (at 300). The agent is therefore never attempting more tactic applications when k increases.

Finally, by combining the best depth limit $d = 10$ and beam width $k = 20$, the agent proves **9.98%** of the theorems in the test set.

Table 5.10: Results for different depth limits d and beam widths k.

d / k	**5**	**10**	**20**
10	-	8.95%	**9.98%**
50	3.94%	8.65%	9.36%
100	-	8.19%	-

5.3.3 Results from Experiment 3

The results from experiment 3 are shown in Table 5.11. Even when exposed to less than 25% of the whole training set, QTac agents generally prove more theorems than the supervised agents. This is encouraging. Moreover, only the C_τ model is trained using deep Q-learning, meaning that there is potentially more to be gained by training argument models in the same way.

Results are similar for both the deep and wide mode. Higher regularization seems to be more critical in the wide mode than the deep. This is unexpected. Recall that the deep mode has QTac attempt the same theorems several times, while the wide mode only once. In other words, the deep model should become more specialized for the theorems it is exposed to. One might expect regularization to be more beneficial in this scenario, but this is not the case. The impact of regularization will likely be clearer if QTac models are trained on more theorems.

Table 5.11: Performance of QTac agents on end-to-end theorem proving. "low" and "medium" refers to the regularization level.

	low	**medium**
Wide	10.33%	10.63%
Deep	**10.74%**	9.51%

Results for Different Coq Projects

Table 5.12 provides an overview of results for each Coq project in CoqGym's test set. The table compares the best GAST and QTac agents, in addition to the random guesser for $d = 10$ and $k = 20$. Note that the `coquelicot` project is marked in red as it was not testable, as explained in Section 5.2.2.

The table shows relatively large differences in how many proofs an agent is able to prove for different Coq projects. For example, projects like `PolTac` and `Demos` seem to be fairly easy, while `verdi` and `verdi-raft` seem to be hard. QTac solves 20.1% of `PolTac` and 67.6% of `demos`, while only solving 6.5% of `verdi-raft` and 7.0% of `verdi`.

In general, QTac beats GAST on most projects. However, there are some projects where

GAST beats QTac by a reasonable margin. For instance, on the project `PolTac`, GAST proves 87 theorems, and QTac proves 73 theorems. This is a relative improvement of 19.18%. A possible explanation for this is that QTac and GAST have learned slightly different proof styles (elaborated more on in Section 6.3.3), where following the QTac proof style is less effective in the `PolTac` project. It could also be because QTac is exposed to fewer proofs than GAST. Perhaps a subset of the training data contains important learning examples for learning to prove the `PolTac` theorems, and QTac might never be exposed to those examples.

Table 5.12: Theorem proving results for different Coq projects. The number of theorems contained in each Coq project is shown next to the project name.

		Random Guesser	**GAST**	**QTac**
`weak-up-to`	139	9 (6.5%)	8 (5.8%)	10 (7.2%)
`buchberger`	725	63 (8.7%)	76 (10.5%)	74 (8.7%)
`jordan-curve-theorem`	628	15 (2.4%)	25 (4.0%)	27 (4.3%)
`dblib`	180	22 (12.2%)	31 (17.2%)	38 (21.1%)
`disel`	634	35 (5.5%)	55 (8.7%)	81 (12.8%)
`zchinese`	43	0 (0.0%)	3 (7.0%)	3 (7.0%)
`zfc`	237	14 (5.9%)	27 (11.4%)	35 (14.8%)
`dep-map`	43	9 (20.9%)	5 (11.6%)	9 (20.9%)
`chinese`	131	15 (11.5%)	18 (13.7%)	30 (22.9%)
`UnifySL`	968	71 (7.3%)	85 (8.8%)	87 (9.0%)
`hoare-tut`	18	0 (0.0%)	0 (0.0%)	2 (11.1%)
`huffman`	314	14 (4.5%)	22 (7.0%)	26 (8.3%)
`PolTac`	363	56 (15.4%)	87 (24.0%)	73 (20.1%)
`angles`	62	3 (4.8%)	4 (6.5%)	4 (6.5%)
`coq-procrastination`	8	2 (25.0%)	2 (25.0%)	3 (37.5%)
`coq-library-undecidability`	2,355	155 (6.6%)	243 (10.3%)	253 (10.7%)
`tree-automata`	828	58 (7.0%)	83 (10.0%)	76 (9.2%)
`fermat4`	130	0 (0.0%)	0 (0.0%)	7 (5.4%)
`demos`	68	43 (63.2%)	49 (72.1%)	46 (67.6%)
`coqoban`	2	0 (0.0%)	0 (0.0%)	0 (0.0%)
`goedel`	606	33 (5.4%)	51 (8.4%)	48 (7.9%)
`verdi-raft`	2,127	75 (3.5%)	119 (5.6%)	139 (6.5%)
`verdi`	514	32 (6.2%)	35 (6.8%)	36 (7.0%)
`zorns-lemma`	149	6 (4.0%)	10 (6.7%)	8 (5.4%)
`coqrel`	256	111 (43.4%)	118 (46.1%)	130 (50.8%)
`fundamental-arithmetics`	142	7 (4.9%)	9 (6.3%)	8 (5.6%)
`coquelicot`	1,467	-	-	-
Total	**11,670**	**848 (7.3%)**	**1,165 (10.0%)**	**1,253 (10.7%)**

Chapter 6

Evaluation and Discussion

The following chapter will evaluate and discuss results from this Master's Thesis. Section 6.1 evaluates and discusses the Research Questions formulated in Chapter 1 in light of the experimental results. Section 6.2 evaluates the Goal formulated in Chapter 1 based on how the Thesis has answered the Research Questions. Section 6.3 discusses interesting findings and relevant topics further.

6.1 Evaluation and Discussion of Research Questions

Research Question 1 *How to design an easy and fast Auto-ITP proxy metric that also indicates end-to-end theorem proving performance?*

The tactic group proxy metric designed in Section 4.3 directly addresses Research Question 1. It is designed to balance the CoqGym dataset and emphasizes overall proof strategy rather than individual tactics. Results from the tactic group experiments (Section 5.3.1) indicate that the BERT-based model BERTac should predict core tactics better than the GCN-based model GAST. This is also the case; comparing the validation scores for the C_τ models from experiment 2 (Table 5.8 in Section 5.3.2), BERTac outperforms GAST by 1.62 percentage points when models are trained on synthetic proofs and by 2.79 percentage points when trained on human-written proofs.

However, this does not directly transfer to improved theorem proving ability as shown by the end-to-end theorem proving accuracies in Table 5.9. In other words, a model performing well on the tactic group experiment is likely to achieve a relatively higher validation score as a C_τ model but is not necessarily a strong model for theorem proving (this is discussed further in Section 6.3.1). A key characteristic of the metric should be that it is indicative of theorem proving performance. Thus, there is still work needed to meet this criterion with the tactic group-based metric proposed in this Thesis.

Another important aspect of the tactic group proxy metric is that it does not address tactic arguments. Although the BERTac model performs better on the tactic group experiment (as noted above), it performs significantly worse as an argument model. This is shown by the BERTac C_{LC} and C_{GC} validation scores in Table 5.8. For example, when trained on synthetic proofs, the BERTac C_{LC} model scores 19.2 percentage points

lower than the GAST C_{LC} model. For the C_{GC} models, the difference is 4.33 percentage points, in GAST's favor. Therefore, a key improvement to the tactic group proxy metric would be to also account for tactic arguments. Another option is to reconsider what the metric should be based on entirely. For example, Huang et al. (2019) propose a metric where the model predicts how many proof steps are left. This metric is neither based on core tactics nor tactic arguments but might still indicate end-to-end theorem proving performance.

Research Question 2 *How can a conceptually simple end-to-end theorem proving agent be designed for tactic-based ITP theorem proving?*

The theorem proving agent designed in Section 4.3 directly addresses Research Question 2. The agent is conceptually simple in that the ITP theorem proving process is interpreted as classic machine learning problems – three separate multi-class classification problems. The three classifiers share the same overall architecture, making model design easier. Furthermore, each classifier is independent of the others meaning models can be combined in arbitrary ways to build a working agent.

The agent is similar to the agents designed for the HOList framework (Bansal et al., 2019a). The primary difference is that this Thesis' agent does not discard the local context and does not consider argument classification as a series of independent binary classification tasks. This makes training argument models less expensive and avoids the problem of the model seeing few positive examples, as explained in Section 4.3.

A drawback of this approach is that the space of potential tactic applications is restricted. The agent designed by Yang and Deng (2019), also used by First et al. (2020), can build tactic applications from the entire Coq tactic space, making them more flexible. Part of the reason why agents in this Thesis are not able to outperform neither ASTactic (Yang and Deng, 2019) nor TacTok (First et al., 2020) could be the limited expressivity of the agent. Expressivity can, however, be improved by further developing the tactic-building module, which can be done independently of the classifiers.

Research Question 3 *What novel embedding techniques can help models perform well in CoqGym?*

Two novel embedding techniques are tested: Graph Convolutional Networks (GCNs) (Kipf and Welling, 2017) and the BERT Transformer architecture (Devlin et al., 2018). Supervised GAST and BERTac agents outperform corresponding random guessing agents. The strongest BERTac agent proves 16.30% more theorems than the corresponding random guesser, and the strongest GAST agent proves 37.28% more theorems than the corresponding random guesser (an overview of main end-to-end theorem proving results can be found in Table 5.6). This indicates that both GCN and off-the-shelf BERT models can be deployed effectively for Auto-ITP in CoqGym. Moreover, when

comparing corresponding GAST and BERTac agents (i.e., when they both use the same depth limit and beam width), the GAST agent proves 8.26% more theorems than the BERTac agent. This can likely be attributed to GAST models significantly outperforming BERTac models on argument prediction as shown in Table 5.8. Results in this Thesis, therefore, suggest that GCN is more suited for Auto-ITP than BERT. Note that tuning hyperparameters was not a significant concern in this Thesis but could provide more nuances to this claim. Some shortcomings of the BERT-implementation in this Thesis will be discussed further in Section 6.3.6. If these are addressed effectively, BERT-based models could perhaps compete with the GCN models.

To gain further insights into Research Question 3, an FFN baseline model, similar to the one developed for the tactic group experiments, would shed more light on the gains made from using GCNs and BERT. Experiment 1 indicates that GAST and BERTac only marginally beat the FFN baseline on core tactic prediction. However, as already explained under Research Question 1, this does not reveal the FFN baseline's ability to predict arguments. It is therefore not obvious how an FFN baseline model would compare to GAST and BERTac models on end-to-end theorem proving.

It is not central to directly compare agents in this Thesis to ASTactic (Yang and Deng, 2019) and TacTok (First et al., 2020) as different theorem proving agents are deployed. An implementation using the same theorem proving agent could provide more insights into how the deep learning models themselves perform. So far, different research groups have designed unique agents, making it difficult to know what performance gains should be contributed to the agent design and what should be contributed to the deep learning models. The same is the case for this Thesis, as a more accessible machine learning interpretation of the theorem proving task was a priority instead of relying on the agent designed by Yang and Deng (2019).

Research Question 4 *How does reinforcement learning compare to supervised learning in CoqGym?*

To answer this Research Question, the deep Q-learning architecture QTac was designed (see Section 4.4.3). QTac leverages the modular theorem proving agent designed in Section 4.3 – it trains a C_τ model and relies on supervised models for argument prediction, making experiments more manageable. QTac generally outperforms the supervised learning agents, as shown in experiment 3 (Section 5.3.3), even when exposed to less than 25% of the CoqGym training data. The best-performing QTac model proves 7.6% more theorems than the best-performing supervised agent. This shows that a deep reinforcement learning method, like deep Q-learning, can be leveraged effectively on the Auto-ITP problem.

A key aspect of QTac is how it generates more training data from existing theorems by learning from both successful and failed proofs. Every intermediate proof step

is a potential learning experience for QTac, and a single explorative proof procedure can generate tens of such proof steps, as explained in Section 5.1.3.

It would be interesting to see how a deep reinforcement learning agent absent of any imitation-style training performs. Note that results from Bansal et al. (2019b) indicate that combining reinforcement learning with imitation training is superior in HOList. Similar results for deep reinforcement learning would likely be the case. Still, it would shed more light on the learning process of deep reinforcement learning Auto-ITP models.

6.2 Evaluation of Goal

The Goal for this Master's Thesis was the following:

Goal *Further progress machine learning applied to formal reasoning by testing new machine learning techniques on the Auto-ITP task.*

Two new deep learning methods have been implemented: A GCN-based and a BERT-based method. GNN-based models have been used in the Auto-ITP context before (Paliwal et al., 2020), but not in CoqGym. However, the related TreeLSTM method has been used in CoqGym (Yang and Deng, 2019; First et al., 2020). GCNs are therefore not an entirely novel approach but rather a natural next step. Applying BERT is more novel. It was inspired by several related works applying Natural Language Processing (NLP) techniques to mathematics and formal reasoning (Rabe et al., 2020; Lample and Charton, 2020; Polu and Sutskever, 2020) (see Section 3.4.1). When trained to imitate human proofs, these techniques outperform corresponding random guessing agents by a large margin – 16.30% for the BERT-based agent and 37.28% for the GCN-based agent.

As part of the work in this Thesis, a new theorem proving agent is developed. This does not directly apply to the Goal of the Thesis but is a step towards better understanding theorem proving from a machine learning perspective. This Master's Thesis argues that this is important, and it was therefore prioritized. A proxy metric was developed, which also does not directly apply to the Goal of the Thesis. However, it allows easier testing of models, which is helpful in the challenging theorem proving domain.

In addition, a deep Q-learning agent was developed and trained using a replay memory. This model further improved results 7.6%, showing that it is possible to use deep reinforcement learning effectively on the Auto-ITP task. This is the first time deep reinforcement learning has been applied to ITP theorem proving. It deals with the problem of data scarcity, which is crucial. This is pointed out by Bansal et al. (2019b) and in the context of theorem synthesis (Wang and Deng, 2020) (see Section 3.4.2). It also lets the agent find a unique proof strategy, effectively dealing with potential noisy human proof styles across different formalization projects (further discussed in Section 6.3.3).

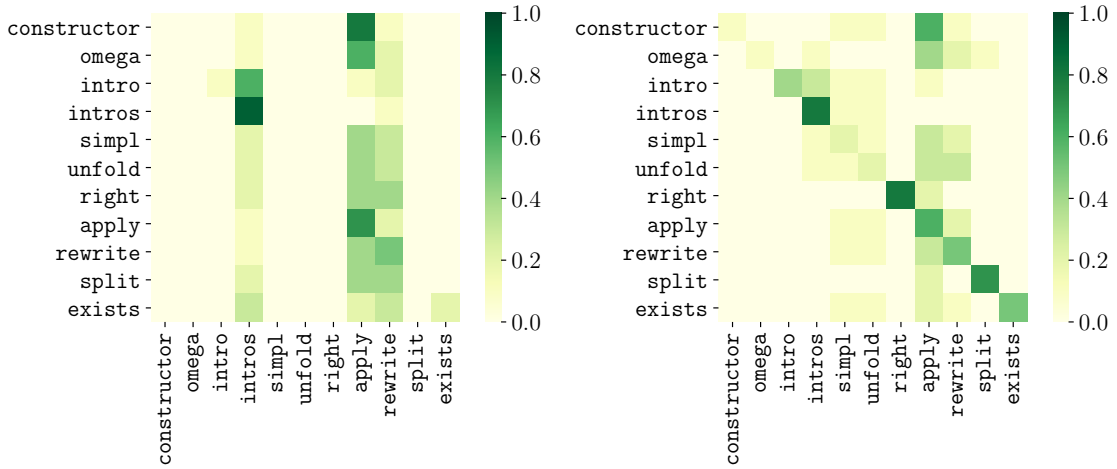
6.3 Further Discussion

Further discussion on findings from experiments and relevant topics now follows. Section 6.3.1 takes a closer look at GAST versus BERTac C_τ models, explaining some of the reason why it does not help to replace the GAST C_τ model with a (higher validation score) BERTac C_τ model (as shown from results in Section 5.3.2). A discussion of the QTac training methodology is included in Section 6.3.2. Proof style – based on core tactic use – is discussed in Section 6.3.3. The CoqGym dataset is further discussed in Section 6.3.4, where the differences between results in CoqGym versus other Auto-ITP frameworks are addressed. Subsection 6.3.5 drills down specifically on the synthetic proof data in CoqGym, discussing why this dataset leads to better results for imitation models than human-written proofs. Transformer models applied to formal logic is discussed in Section 6.3.6 where some of the shortcomings of the BERTac models are brought up. Finally, Section 6.3.7 notes some comparisons to Hammers and Section 6.3.8 briefly discusses proof tree traversal.

6.3.1 C_τ Predictions

To better understand the predictions made by supervised learning models, confusion matrices for the weakest and strongest supervised C_τ models are plotted. Figure 6.1a shows the confusion matrix for the GAST C_τ model trained on human-written proofs. Figure 6.1b shows the confusion matrix for the BERTac C_τ model trained on synthetic proofs. These are built from the predictions made during validation. Only a subset of core tactics is included for the sake of readability. See Section 2.2.5 for an explanation of relevant tactics.

Both models are biased towards predicting `apply` and `rewrite`. This indicates that the C_τ models are dependent on effective C_{GC} models in order to function well, as both `apply` and `rewrite` are dependent on arguments from the global context to work with the theorem proving agent. This is especially true for the GAST model. Note that the proof step datasets are unbalanced (explained in Section 4.2), meaning the strong bias towards predicting only a few tactics is expected and still allows the models to achieve a validation score greater than 30% (as shown in Table 5.8). The BERTac model predicts a wider span of tactics. This could explain the reasons why end-to-end theorem proving performance does not necessarily improve when changing from a pure GAST-based agent to a combination where C_τ is a BERTac model (shown in Table 5.9). Even though C_τ has a strong validation score, it is not going to help the agent if it proposes tactics that are not effective in driving theorem proving process forward. For instance, BERTac C_τ is able to predict the tactic `split` reasonably well, which is not the case for the GAST C_τ model, as shown in the confusion matrices. This leads to a higher validation score but is not necessarily helpful when proving theorems, as it simply splits subgoal into two new subgoals. `split` is also not a common tactic in human proofs – it is used only in 2.4% of human-written proofs. This is much less than more popular tactics like `apply` and `rewrite` (used in 19.47% and 11.07% of human proofs, respectively).



(a) Confusion matrix for the GAST C_τ model trained on human-written proofs.

(b) Confusion matrix for the BERTac C_τ model trained on synthetic proofs.

Figure 6.1: Confusion matrices for C_τ models. The true tactic is denoted along the vertical axis and the predicted along the horizontal. Values are normalized to a probability between zero and one. Only a subset of tactics are included, meaning each row does not necessarily add up to one.

The importance of the C_{GC} model, as explained above, is interesting. Predicting global context arguments is essentially the same as the premise selection problem (explained in the context of Hammers in Section 3.3.2). This problem has been pointed out as a critical part of theorem proving in several contexts (Hoder and Voronkov, 2011; Gauthier and Kaliszyk, 2015; Wang et al., 2017) (for example in Hammers (Gauthier and Kaliszyk, 2015) and traditional ATP systems (Hoder and Voronkov, 2011)). This is, as shown here, also a critical problem for the Auto-ITP agents.

6.3.2 QTac Training

One of the main advantages of deploying deep reinforcement learning for Auto-ITP is that it deals with data scarcity. This has been pointed out as a key bottleneck in theorem proving (Wang and Deng, 2020). The QTac model in this Thesis tackles this because it learns from not just successful proofs, but also failed proofs. As explained in Section 5.1.3, it generates around 500k proof step experiences from 10k proof attempts (less than 25% of CoqGym's training set). This is much more data than the 296k proof steps in the combined CoqGym human-written and synthetic datasets. Of course, with many failed proof attempts, there is likely to be lots of noise in the replay memory dataset. To deal with this, several techniques can be applied. QTac uses a target network to stabilize training (Mnih et al., 2015), only trains on a subset of the replay memory with a guarantee that successful proofs will be part of the subset (as explained in Section 4.4.3).

However, it is not clear from experiments exactly how QTac responds to different training methods. For instance, applying more regularization to the model does not yield any conclusive indications for how this affects training. Furthermore, exposing the agent to fewer proofs more times (i.e., the "deep" mode, see Section 4.4.3) versus more proofs one time (i.e., the "wide" mode, see Section 4.4.3) also does not conclusively indicate which one is preferable as results are similar (see Table 5.11 for QTac results). Time constraints meant that more QTac experiments had to be left out, leaving the above questions for future study.

6.3.3 Proof Style

A way to get more insights into how agents prove theorems is by looking at the frequency of how often they deploy different tactics. Figure 6.2 plots tactic frequency for successful proofs, for the GAST and QTac agents and human-written Coq proofs. The random guessing agent is also included for comparison. Only a subset of the most common tactics are included, for the sake of readability. See Section 2.2.5 for an explanation of important tactics. The following observations can be made.

The random guesser relies primarily on tactics that do not use arguments. Instead, it proves theorem by leveraging Coq's internal automatic engines. For the random guesser to successfully include arguments, it would have to guess both the core tactic and the argument correctly, which is difficult.

Both the GAST and QTac agents make fair use of the internal automatic engine `auto`. This is expected, as `auto` is a powerful tactic capable of proving non-trivial subgoals automatically. Yang et al. (2016) report that `auto` can prove 2.9% of the whole CoqGym test set by itself. Such engines are used less often in human proofs.

Both the GAST and QTac agent uses `intro` and `intros` actively. This is expected as `intros` is a popular tactic also among human proofs. In addition, the agents rely on local hypotheses to supply local context arguments, not direct references to terms contained in subgoals. This does not hinder the agent, as was mentioned in Section 2.2.2, as long as the local context is modified so that subgoal terms are part of the local hypotheses. This is achieved by using `intro` and `intros`.

The QTac agent uses `induction` 137.42% more often than the GAST agent. This is interesting because `induction` is a tactic dependent on arguments from the local context to work (as shown in Table 2.1). The C_{LC} model used by QTac has a significantly higher validation than the C_{GC} it uses (shown in Table 5.8). A weak C_{GC} model leads to tactics dependent on arguments from the global context (e.g., `apply` or `rewrite`) to fail more often, likely making QTac tend towards the local context-dependent tactic `induction` instead. In other words, QTac appears to adopt its proof style, in a way not possible for imitation-based agents.

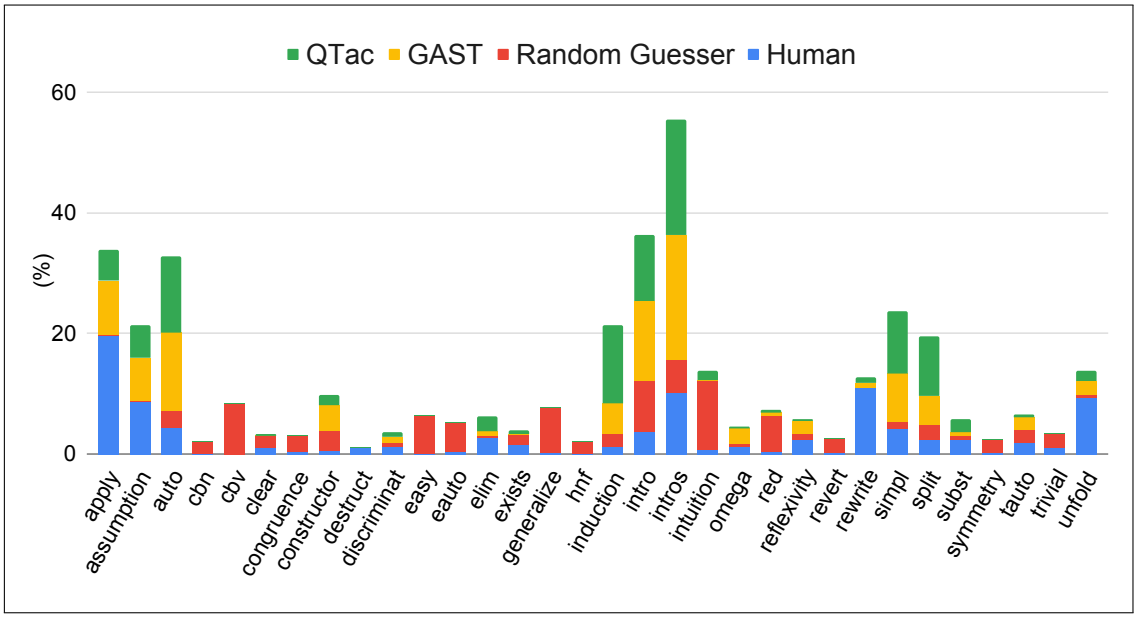


Figure 6.2: Frequency of core tactic use for different proof agents. The plot is stacked for improved readability.

6.3.4 The CoqGym Dataset

Results in CoqGym may in part be explained by the way the CoqGym dataset is split between train, validation, and test sets. It is important to realize that the theorems in the different splits come from different Coq projects. Models, therefore, need to learn how to prove theorems independent of Coq projects to be able to generalize well to the test set. However, this can be difficult as many aspects of Coq projects differ. Yang and Deng (2019) point out that the average number of theorems in the global context varies significantly across different projects. For instance, in the `CompCert` the average number is 13,340 and in `InfSeqExt` it is 661 (Yang and Deng, 2019). Table 5.12 also supports this, as it shows how performance varies significantly across different Coq projects. For example, the best-performing *Q*Tac agent can prove 50.78% of the `coqrel` project and only 6.54% of the `verdi-raft` project.

Perhaps it would be more reasonable to make the Auto-ITP task easier by focusing exclusively on one project. The model would train on a subset of theorems from that specific project before being tested on the same project. In this context, the problem of *data leakage* has to be addressed. In the Hammer context, this is dealt with by *human-chronological* corpus building (described in Section 3.3.1) (Blanchette et al., 2016). The performance difference between models in CoqGym and other Auto-ITP frameworks, like HOList, might also be explained this way. The datasets in HOList are sorted in a human-chronological ordering (Bansal et al., 2019a), meaning that generalization from training projects to test projects might be less of an issue in this

framework. Results between different Auto-ITP are therefore hard to compare directly.

Tactics dependent on arguments from the global context are some of the most popular tactics in human-written proofs. For example, `apply`, `rewrite` and `unfold` are collectively used in 39.94% of the human proofs in the training set (see Table 4.1 for an overview of tactic frequency in human proofs). The average number of theorems in the global context across all projects is 10,350.3 (Yang and Deng, 2019), more than one hundred times more than the ten theorems included in global context supplied to ASTactic, TacTok, and the agents in this Thesis. This global context restriction is likely to be a significant.

6.3.5 CoqGym's Synthetic Data

As explained in Section 3.2.4, CoqGym ships with synthetic data extracted from human proofs. Results from end-to-end theorem proving experiments in Section 5.3.2 show that this dataset increases performance over human-written proofs. However, it is not apparent exactly why this is. It is important to note, as also noted by Yang and Deng (2019), that because the synthetic proof data relies on tactics extracted from human proofs, it should not lead to a radically different theorem proving strategy than human-written proofs. The data does not serve as a replacement for reinforcement learning but instead provides more human-like labeled training data in CoqGym.

However, the unique characteristics of the synthetic proofs can provide insights into how this data affects models. Synthetic proofs finish proofs by using the `auto` tactic. This is interesting. It could be that this leads to less noise when training models because the data is consistent about how to solve simple subgoals. Moreover, synthetic proofs start by moving subgoals to the list of local hypotheses. This leads to a larger space of potential local context arguments (Figure 5.1 explicitly shows this). Perhaps this is why C_{LC} models (as the only model) perform better when trained on human-written proofs (shown in Table 5.8).

6.3.6 Tailoring Transformer Models to Formal Expressions

Formal expressions are not the same as natural human language. This clearly shows when comparing pre-trained versions of BERTac to one without pre-trained weights (see validation plots in Figure 5.5). The pre-trained versions perform worse across the board, meaning that NLP-tailored pre-training seems not to transfer well to Coq expressions. Pre-training tailored to formal expressions – similar to the he skip-tree task (Rabe et al., 2020) and the GPT-f system (Polu and Sutskever, 2020) – is a way to address this.

When deploying the model as an argument classifier (in which multiple expressions are handled), the model performs significantly lower than the GCN models. This points to some potential weak points of the BERTac model. An off-the-shelf separation token might not be well-suited for Auto-ITP. It could also be a problem

for BERTac that the concatenation of several Coq expressions results in sequences too large for the BERT model. The preprocessing step where identifiers and expressions are concatenated (explained in Section 4.4.2) could be modified too, by introducing a designated BERT-style token for this purpose.

6.3.7 Comparison to Hammers

Hammers (described in Section 3.3) are a radically different way of automating ITP systems. So far, CoqHammer (Czajka and Kaliszyk, 2018) significantly outperforms Auto-ITP models in Coq. Yang and Deng (2019) report that CoqHammer is able to prove 24.8% of CoqGym's test set. This is 11.9 percentage points more than TacTok and 14.06 more than the best-performing *Q*Tac agent. In other words, highly optimized ATP systems, using classical inference techniques (explained in Section 2.1), prove hard to outperform in Coq theorem proving for now. However, results can not be compared directly as CoqHammer deploys a premise selection step using machine learning models trained in a human-chronological way, not using CoqGym's training data. It could even be that part of CoqHammer's premise selection training data overlaps with CoqGym's test data.

Integration between Hammers and Auto-ITP models have significantly boosted results in CoqGym, with ASTactic improving results by 17.8 percentage points when CoqHammer calls are interleaved with tactic prediction (Yang and Deng, 2019). A similar integration for agents in this Thesis is possible but was not the focus of the Thesis. It would be interesting to test, as it allows further investigation into proof style. In particular, it would reveal the overlap between what theorems CoqHammer and Auto-ITP agents can prove. Perhaps, for example, *Q*Tac overlaps more with CoqHammer than supervised agents and would therefore benefit less from integrated Hammer calls.

6.3.8 Proof Tree Traversal

An interesting topic not focused on in this Thesis is proof tree traversal. Agents deploy Depth-First Search following Yang and Deng (2019), and First et al. (2020). In HOList, Breath-First Search has been used (Bansal et al., 2019a) and TacticToe agents have been equipped with more sophisticated heuristic-based strategies – one based on A* (Gauthier et al., 2017) and one based on Monte Carlo tree search (Gauthier et al., 2020). This latter modification resulted in significant improvements in TacticToe (explained in Section 3.2.1), indicating that proof tree traversal has an impact on performance. Furthermore, Yang and Deng (2019) point out that ASTactic typically finds much shorter proofs than typical human-written proofs. This, and the fact that a lower depth limit in experiment 2 (Section 5.3.2) improves accuracy, indicates that traversing far down a branch in the proof tree is not desirable. Therefore, using a Breath-First Search, like in HOList, could potentially improve results in CoqGym.

Chapter 7

Conclusion and Future Work

To conclude the Master's Thesis, Section 7.1 summarizes contributions and Section 7.2 ends with some notes on possible avenues for future research.

7.1 Contributions

This Master's Thesis designs a new proxy metric for the CoqGym framework and argues why such metrics are helpful for the theorem proving domain. The proxy metric is based on grouping related tactics together into *tactic groups* (see Section 4.2). The grouping allows the tactic dataset to become more balanced and emphasizes proof strategy rather than specific tactics. This metric can provide a step towards easier prototyping of Auto-ITP models. Further improvements will be to also include tactic arguments in the proxy metric.

A new theorem proving agent is designed for Interactive Theorem Proving (ITP) (Section 4.3). This agent turns the ITP proof procedure into three separate multi-class classification problems. Each classification problem focuses on one of three key aspects of tactic applications – the core tactic, the local context, and the global context. This provides a natural machine learning interpretation of the proof procedure, making it suited for machine learning research. Building tactics based on the output from each classifier is done in a separate module and can be tailored (e.g., building tactics consisting of more than one argument) independently. This agent is not unique to Coq, as most ITP systems implement an almost identical proof procedure with core tactics and a local and global context. Furthermore, each classifier operates independently of the others meaning classifiers can be combined in any arbitrary way.

Experiments in this Thesis focus on two deep learning embedding techniques. One is a Graph Neural Network (GNN) technique based on Graph Convolutional Networks (GCN) (Kipf and Welling, 2017) and the end-to-end graph classification architecture DGCNN (Zhang et al., 2018). This implementation uses GCN message passing to obtain node embeddings of the Abstract Syntax Tree (AST) representations of Coq expressions before pooling node embeddings to obtain a graph representation (see Figure 4.5 for the model architecture). The other is the Transformer model BERT (Devlin et al., 2018). This model obtains embeddings by leveraging self-attention

techniques directly on Coq expressions (see Figure 4.6 for the model architecture). Furthermore, BERT is tested with and without pre-trained weights. Pre-trained weights are obtained from Natural Language Processing (NLP) tasks, and results show that this pre-training process does not transfer well to formal expressions (shown in Section 5.3.2). This is the first time GCNs and Transformers have been used for tactic-based ITP theorem proving.

Several combinations of supervised GCN and BERT models are tested for end-to-end theorem proving (results are presented in Section 5.3.2). The GCN models outperform the BERT models. This can be attributed to GCN models working significantly better as argument classifiers. Several proof tree depth limits and beam widths are tested. Results are consistent with previous results (Yang and Deng, 2019; First et al., 2020): lowering the depth limit to 10 and increasing the beam width to 20 improves results.

A deep reinforcement learning agent is developed for CoqGym (see Figure 4.7 for the agent architecture). This agent is based on deep Q-learning (Mnih et al., 2015) and trains by interleaving replay memory training with imitation training. The agent trains a core tactic classifier and relies on supervised models for argument prediction. Analysis of the deep Q-learning agent's proof style in Section 6.3 reveals that it relies more on the `induction` tactic supervised learning agents. This is interesting as the `induction` tactic is dependent on arguments from the local context to work, and the local context classifier paired with the deep Q-learning agent is stronger than the global context classifier. This indicates that the deep Q-learning agent adapts its proof style so that it leverages its strongest argument model. Furthermore, it trains on around 400k proof steps generated from less than 25% of CoqGym's test set showing how a deep reinforcement learning approach can tackle the problem of data scarcity, as discussed in Section 6.3.2. Results presented in Section 5.3.3 show that this agent is capable of proving 7.55% more theorems than corresponding supervised agents.

The best-performing agent in this Master's Thesis scored 2.16 percentage points lower than state-of-the-art (First et al., 2020). However, results are difficult to compare directly as a new agent was deployed in this Thesis. As a more appropriate comparison, a random guessing baseline agent was tested. All agents prove significantly more theorems than corresponding random guessing agents: BERT-based agent 16.30% more, GCN-based agent 37.28% more, and deep Q-learning agent 47.73% more. As an interesting side note, best-performing BERT, GCN, and deep Q-learning agents prove 63.06%, 103.67%, and 119.18% more theorems than Coq's best-performing internal automatic engine `easy` (Yang and Deng, 2019), respectively.

7.2 Future Work

Several directions for future work are possible and will be summarized here. Some directly build on work in this Thesis, some are inspired by work in this Thesis, while other are

completely different avenues to explore.

Further Exploration of Deep Reinforcement Learning

A bottleneck for training machine learning models on formal proof data is the scarcity of labeled data. This hinders supervised learning. A way to generate more training data from theorems is by using reinforcement learning. This has been explored in this Thesis and by others (Wu et al., 2020; Bansal et al., 2019b). An effective theorem proving model will likely leverage a combination of supervised and reinforcement training and can be an exciting path to explore further. Many directions are possible. A potential starting point is to apply deep reinforcement learning, which has shown promising initial results in this Thesis, to not just core tactic prediction but also argument prediction. Another is to explore the balance between exploration and exploitation further. In this context, the ideas of training the model in a "deep" vs. "wide" mode, as described in Section 4.4.3, could be better understood. Also, training models on more proofs can be tested and will lead to a better understanding of how regularization affects the model. Another way to gain more insights into the deep reinforcement learning process is to gather more statistics during training. For instance, a sliding average over the last n proof attempts could reveal more about the convergence rate of different models.

Focus on Premise Selection in the Auto-ITP Context

Analysis of theorem proving agents and work by others (Hoder and Voronkov, 2011; Gauthier and Kaliszyk, 2015; Wang et al., 2017) point to premise selection as a critical aspect of theorem proving. For Auto-ITP, this can be interpreted as global context classification. This is also an essential aspect of Hammers (Gauthier and Kaliszyk, 2015) and traditional ATP systems (Hoder and Voronkov, 2011), making it an essential topic in its own right. A new way to study this problem is as a standalone problem in the Auto-ITP context. For example, one can define the only available core tactic to be anything similar to Coq's `apply` tactic and focus on designing the best premise selection model for this tactic. This can be deployed as a simple Auto-ITP agent focused solely on premise selection.

Further Developing Proxy Metrics and Theorem Proving Agents

This Master's Thesis argues that the development of theorem proving proxy metrics can benefit Auto-ITP research. Such metrics need to account for both core tactics and tactic arguments. Further developing standardized theorem proving agents can also be a valuable topic for future research, as an even more unambiguous machine learning interpretation of the theorem proving task is desirable.

Further developing the agent proposed in this Thesis is a possible starting point. One concrete way to improve this agent is to implement the tactic application ranking scheme used in Proverbot9001 (Sanchez-Stern et al., 2020). Sanchez-Stern et al. (2020) show

that it is beneficial to combine the ranking of core tactics and tactic arguments before deciding on a tactic application. A similar idea can be pursued for this Thesis' agent. Given its modular design, this will be a reasonably straightforward customization to make. Another concrete way to further develop the agent is to build tactics containing multiple arguments from both the local and global context.

Improvements on BERT and Other Transformer Models Applied to Auto-ITP

Several improvements on the BERT-based model are possible as this Thesis only deployed an off-the-shelf solution. Some key topics here (as discussed in Section 6.3.6) are tokenization, handling multiple expressions, dealing with expression identifiers, and pre-training. Furthermore, other Transformer models can be explored too. Several related works focus on Transformer-based models tailored to formal logic and mathematics (Rabe et al., 2020; Lample and Charton, 2020; Polu and Sutskever, 2020) and can be built on for future research along these lines.

Integration with Hammers

Integration between Hammers (Blanchette et al., 2016) and Auto-ITP models is yet another interesting topic. Yang et al. (2016) show how calls to CoqHammer (Czajka and Kaliszyk, 2018) significantly boost results. On the other hand, Hammer calls are expensive, as pointed out by Gauthier et al. (2017) in the context of the TacticToe system. Understanding how to best leverage both Hammers and Auto-ITP models is therefore relevant. One option for future work in this direction is to develop an agent leveraging meta-classifiers responsible for deciding when to call the Hammer versus the Auto-ITP model. Hammers are also interesting because they can be used to compare the overlap of theorems classic inference techniques (i.e., the ATP inference) and tactic-based machine learning models (i.e., the Auto-ITP models) solve. It would be interesting to investigate this overlap for both imitation-style Auto-ITP agents and deep reinforcement learning agents to understand better the difference between proof styles in these two settings.

Unified Benchmarks and Frameworks

Auto-ITP research (and machine learning applied to formal mathematics at large) has been studied in the context of several systems. Some have focused on HOL4 (Gauthier et al., 2017, 2020), some on HOL Light (Bansal et al., 2019a; Paliwal et al., 2020; Bansal et al., 2019b), and some on Coq (Yang and Deng, 2019; Huang et al., 2019; First et al., 2020; Sanchez-Stern et al., 2020). Similar work has also been based on the MetaMath system (Wang et al., 2017; Polu and Sutskever, 2020). Even in parallel to the writing of this Master's Thesis, two new Auto-ITP frameworks have been proposed: LeanStep (Han et al., 2021) and IsarStep (Li et al., 2021)[1]. While this is encouraging, it also makes it harder to compare models and performance. A unified system and benchmark could

[1]This work is not mentioned in Chapter 3, as it was published in parallel to writing the Master's Thesis.

therefore be useful. Very recently, such work has started to take place, with OpenAI working on a common benchmark across different formal systems[2]. This is promising for the field and an exciting avenue for further research. As discussed in Section 6.3.4, important considerations for such a benchmark is human-chronological versus more traditional train-validation-test splits, as mentioned in Section 6.3.4.

Combining Autoformalization and Formal Reasoning

Another fascinating topic is described by Szegedy (2020). Szegedy (2020) proposes autoformalization as a natural extension of machine learning-based formal reasoning. Most information is not stated formally but rather informally. An autoformalization system would be able to map informal information to formal expressions. Furthermore, one can imagine an end-to-end system, where an autoformalization module takes in informal text, maps it to logical expressions, and an Auto-ITP-like system can reason over the formal expressions effectively. The output of the formal reasoning procedure could then be mapped back to informal human-readable information. As argued by Szegedy (2020), such an end-to-end system would combine strong NLP models with formal reasoning capabilities.

[2]https://github.com/openai/miniF2F

Bibliography

Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise Selection for Mathematics by Corpus Analysis and Kernel Methods. *Journal of Automated Reasoning*, 52(2):191–213, 2014.

Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pąk. The Role of the Mizar Mathematical Library for Interactive Proof Development in Mizar. *Journal of Automated Reasoning*, 61:9–32, 2018.

Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. HOList: An Environment for Machine Learning of Higher-Order Theorem Proving. In *International Conference on Machine Learning*, pages 454–463, 2019a.

Kshitij Bansal, Sarah M Loos, Markus N Rabe, and Christian Szegedy. Learning to Reason in Large Theories without Imitation. *arXiv preprint arXiv:1905.10501*, 2019b.

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Technical report, INRIA, 1997.

Wolfgang Bibel. Early History and Perspectives of Automated Deduction. In *Annual Conference on Artificial Intelligence*, pages 2–18. Springer, 2007.

Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C Paulson, and Josef Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.

Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement Day. In *International Joint Conference on Automated Reasoning*, pages 107–121. Springer, 2010.

Robert Boyer. The QED Manifesto. *Automated Deduction - CADE*, 12:238–251, 1994.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37, 2015.

Adam James Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

Alonzo Church. A note on the Entscheidungsproblem. *The Journal of Symbolic Logic*, 1 (1):40–41, 1936.

Sylvain Conchon and Jean-Christophe Filliâtre. A Persistent Union-Find Data Structure. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 37–46, 2007.

Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning*, 61(1-4):423–453, 2018.

Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962.

David Delahaye. A Tactic Language for the System Coq. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 85–95. Springer, 2000.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE, 2009.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

William Ewald. The Emergence of First-Order Logic. In *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2019 edition, 2019.

Matthias Fey and Jan E. Lenssen. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Emily First, Yuriy Brun, and Arjun Guha. TacTok: Semantics-Aware Proof Synthesis. *Proceedings of the ACM on Programming Languages*, 4:1–31, 2020.

Emilio Jesús Gallego Arias. SerAPI: Machine-Friendly, Data-Centric Serialization for Coq. Technical report, MINES ParisTech, 2016.

Thibault Gauthier and Cezary Kaliszyk. Premise Selection and External Provers for HOL4. In *Conference on Certified Programs and Proofs*, pages 49–57, 2015.

Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. TacticToe: Learning to Reason with HOL4 Tactics. In *21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 125–143. EasyChair, 2017.

Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. TacticToe: Learning to Prove with Tactics. *Journal of Automated Reasoning*, pages 1–30, 2020.

Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

Georges Gonthier. Formal Proof–The Four-Color Theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Roux, Assia Mahboubi, Russell O'Connor, Sidi Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.

Mike Gordon. From LCF to HOL: A Short History. In *Proof, language, and interaction*, pages 169–186, 2000.

Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a Nutshell. *Journal of Formalized Reasoning*, 3(2):153–245, 2010.

Thomas Hales, Mark Adams, Gertrud Bauer, Dat Dang, John Harrison, Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Nguyen, Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Ta, Trân Trung, Diep Trieu, and Roland Zumkeller. A formal proof of the Kepler conjecture. In *Forum of mathematics, Pi*, volume 5. Cambridge University Press, 2017.

Thomas C Hales. Introduction to the Flyspeck Project. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W Ayers, and Stanislas Polu. Proof Artifact Co-training for Theorem Proving with Language Models. *arXiv preprint arXiv:2102.06203*, 2021.

John Harrison. HOL Light: A tutorial introduction. In *International Conference on Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996.

John Harrison. Floating Point Verification in HOL Light: The Exponential Function. *Formal Methods in System Design*, 16(3):271–305, 2000.

John Harrison. HOL Light: An overview. In *International Conference on Theorem Proving in Higher Order Logics*, pages 60–66. Springer, 2009.

John Harrison, Josef Urban, and Freek Wiedijk. History of Interactive Theorem Proving. In *Computational Logic*, volume 9, pages 135–214, 2014.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural computation*, 9(8):1735–1780, 1997.

Kryštof Hoder and Andrei Voronkov. Sine Qua Non for Large Theory Reasoning. In *International Conference on Automated Deduction*, pages 299–314. Springer, 2011.

Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. GamePad: A Learning Environment for Theorem Proving. In *International Conference on Learning Representations*, 2019.

Joe Hurd. First-Order Proof Tactics in Higher-Order Logic Theorem Provers. *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68, 2003.

Cezary Kaliszyk and Josef Urban. Stronger Automation for Flyspeck by Feature Weighting and Strategy Evolution. In *Third International Workshop on Proof Exchange for Theorem Proving*, volume 14 of *EPiC Series in Computing*, pages 87–95. EasyChair, 2013.

Cezary Kaliszyk and Josef Urban. Learning-Assisted Automated Reasoning with Flyspeck. *Journal of Automated Reasoning*, 53(2):173–213, 2014.

Cezary Kaliszyk and Josef Urban. FEMaLeCoP: Fairly Efficient Machine Learning Connection Prover. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 88–96. Springer, 2015a.

Cezary Kaliszyk and Josef Urban. HOL(y)Hammer: Online ATP service for HOL Light. *Mathematics in Computer Science*, 9(1):5–22, 2015b.

Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *Journal of Automated Reasoning*, 55(3):245–256, 2015c.

Cezary Kaliszyk, François Chollet, and Christian Szegedy. HolStep: A Machine Learning Dataset for Higher-order Logic Theorem Proving. *International Conference on Learning Representations*, 2017.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*, 2017.

Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. *International Conference on Learning Representations*, 2017.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems*, 32(1):1–70, 2014.

Donald E. Knuth and Peter B. Bendix. Simple Word Problems in Universal Algebras. In *Computational Problems in Abstract Algebra*, pages 263 – 297. Pergamon, 1970.

Laura Kovács and Andrei Voronkov. First-Order Theorem proving And Vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

Anders Krogh and John A Hertz. A Simple Weight Decay Can Improve Generalization. In *Advances in neural information processing systems*, pages 950–957, 1992.

Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. Overview and Evaluation of Premise Selection Techniques for Large Theory Mathematics. In *International Joint Conference on Automated Reasoning*, pages 378–392. Springer, 2012.

Guillaume Lample and François Charton. Deep Learning for Symbolic Mathematics. *International Conference on Learning Representations*, 2020.

Dennis Lee, Christian Szegedy, Markus Rabe, Sarah Loos, and Kshitij Bansal. Mathematical Reasoning in Latent Space. In *International Conference on Learning Representations*, 2020.

Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.

Xavier Leroy. *The CompCert C verified compiler: Documentation and user's manual*. PhD thesis, Inria, 2016.

Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C Paulson. IsarStep: a Benchmark for High-level Mathematical Reasoning. In *International Conference on Learning Representations*, 2021.

Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. *arXiv preprint arXiv:1701.06972*, 2017.

William McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

Norman Megill and David A Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. Lulu.com, 2019.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

M Saqib Nawaz, Moin Malik, Yi Li, Meng Sun, and M Lali. A Survey on Theorem Provers in Formal Methods. *arXiv preprint arXiv:1912.03028*, 2019.

M Saqib Nawaz, M Zohaib Nawaz, Osman Hasan, Philippe Fournier-Viger, and Meng Sun. Proof searching and prediction in HOL4 with evolutionary/heuristic and deep learning techniques. *Applied Intelligence*, pages 1–22, 2020.

Michael A Nielsen. *Neural Networks and Deep Learning*, volume 25. Determination press San Francisco, CA, 2015.

Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation Functions: Comparison of Trends in Practice and Research for Deep Learning. *International Conference on Computational Sciences and Technology*, 2021.

Aditya Paliwal, Sarah M Loos, Markus N Rabe, Kshitij Bansal, and Christian Szegedy. Graph Representations for Higher-Order Logic and Theorem Proving. In *Association for the Advancement of Artificial Intelligence*, pages 2967–2974, 2020.

Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *International Conference on Acoustics, Speech and Signal Processing*, pages 5206–5210. IEEE, 2015.

Peter J. Huber. Robust Estimation of a Location Parameter. *The Annals of Mathematical Statistics*, 35(1):73 – 101, 1964.

Stanislas Polu and Ilya Sutskever. Generative Language Modeling for Automated Theorem Proving. *arXiv preprint arXiv:2009.03393*, 2020.

Markus N Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. Mathematical Reasoning via Self-supervised Skip-tree Training. *arXiv preprint arXiv:2006.04757*, 2020.

Samik Raychaudhuri. Introduction to Monte Carlo simulation. In *2008 Winter simulation conference*, pages 91–100. IEEE, 2008.

Alan JA Robinson and Andrei Voronkov. *Handbook of Automated Reasoning*, volume 1. Elsevier, 2001.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

Michael Rusinowitch. Theorem-proving with Resolution and Superposition. *Journal of Symbolic Computation*, 11:21–49, 1991.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 3rd edition, 2010.

Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. Generating Correctness Proofs with Neural Networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 1–10, 2020.

Stephan Schulz. E - A Brainiac Theorem Prover. *AI Communications*, 15, 09 2002.

Burr Settles. Active Learning Literature Survey. 2009. URL `http://active-learning.net/`.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144, 2018.

Magnus Själander, Magnus Jahre, Gunnar Tufte, and Nico Reissmann. EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure, 2019.

Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't Decay the Learning Rate, Increase the Batch Size. *International Conference on Learning Representations*, 2018.

Raymond Smullyan. *First-Order Logic*. Springer, 1968.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Christian Szegedy. A Promising Path Towards Autoformalization and General Artificial Intelligence. In *International Conference on Intelligent Computer Mathematics*, pages 3–20. Springer, 2020.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, 2015.

Alan Mathison Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230–265, 1936.

Josef Urban and Jiří Vyskočil. Theorem Proving in Large Formal Mathematics as an Emerging AI Field. In *Automated Reasoning and Mathematics*, pages 240–257. Springer, 2013.

Josef Urban, Jiří Vyskočil, and Petr Štěpánek. MaLeCoP Machine Learning Connection Prover. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 263–277. Springer, 2011.

Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. WaveNet: A Generative Model for Raw Audio. In *9th ISCA Speech Synthesis Workshop*, pages 125–125, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Jouko Väänänen. Second-order and Higher-order Logic. In *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall edition, 2020.

Mingzhe Wang and Jia Deng. Learning to Prove Theorems by Learning to Generate Theorems. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise Selection for Theorem Proving by Deep Graph Embedding. In *Advances in Neural Information Processing Systems*, pages 2786–2796, 2017.

Boris Weisfeiler and A. A. Lehmann. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, pages 12–16, 1968.

Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying Graph Convolutional Networks. In *International conference on machine learning*, pages 6861–6871, 2019.

Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. Reinforcement Learning for Interactive Theorem Proving in HOL4. *5th Conference on Artificial Intelligence and Theorem Proving*, 2020.

Kaiyu Yang and Jia Deng. Learning to Prove Theorems via Interacting with Proof Assistants. In *International Conference on Machine Learning*, pages 6984–6994, 2019.

Li-An Yang, Jui-Pin Liu, Chao-Hong Chen, and Ying-ping Chen. Automatically Proving Mathematical Theorems with Evolutionary Algorithms and Proof Assistants. In *Congress on Evolutionary Computation*, pages 4421–4428. IEEE, 2016.

Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An End-to-End Deep Learning Architecture for Graph Classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Magnus Midtbø Kristiansen

Proving Theorems Using Deep Learning

NTNU
Norwegian University of
Science and Technology